A Practical Guide to Randomized Matrix Computations with MATLAB Implementations[1]

Shusen Wang
wssatzju@gmail.com

November 4, 2015

[1]Sample MATLAB code with demos is available at https://github.com/wangshusen/RandMatrixMatlab.

Contents

Abstract

Matrix operations such as matrix inversion, eigenvalue decomposition, singular value decomposition are ubiquitous in real-world applications. Unfortunately, many of these matrix operations so time and memory expensive that they are prohibitive when the scale of data is large. In real-world applications, since the data themselves are noisy, machine-precision matrix operations are not necessary at all, and one can sacrifice a reasonable amount of accuracy for computational efficiency.

In recent years, a bunch of randomized algorithms have been devised to make matrix computations more scalable. Mahoney [16] and Woodruff [34] have written excellent but very technical reviews of the randomized algorithms. Differently, the focus of this paper is on intuition, algorithm derivation, and implementation. This paper should be accessible to people with knowledge in elementary matrix algebra but unfamiliar with randomized matrix computations. The algorithms introduced in this paper are all summarized in a user-friendly way, and they can be implemented in lines of MATLAB code. The readers can easily follow the implementations even if they do not understand the maths and algorithms.

Keywords: matrix computation, randomized algorithms, matrix sketching, random projection, random selection, least squares regression, randomized SVD, matrix inversion, eigenvalue decomposition, kernel approximation, the Nyström method.

Chapter 1

Introduction

Matrix computation plays a key role in modern data science. However, matrix computations such as matrix inversion, eigenvalue decomposition, SVD, etc, are very time and memory expensive, which limits their scalability and applications. To make large-scale matrix computation possible, randomized matrix approximation techniques have been proposed and widely applied. Especially in the past decade, remarkable progresses in randomized numerical linear algebra has been made, and now large-scale matrix computations are no longer impossible tasks.

This paper reviews the most recent progresses of randomized matrix computation. The papers written by Mahoney [16] and Woodruff [34] provide comprehensive and rigorous reviews of the randomized matrix computation algorithms. However, their focus are on the theoretical properties and error analysis techniques, and readers unfamiliar with randomized numerical linear algebra can have difficulty when implementing their algorithms.

Differently, the focus of this paper is on intuitions and implementations, and the target readers are those who are familiar with basic matrix algebra but has little knowledge in randomized matrix computations. All the algorithms in this paper are described in a user-friend way. This paper also provides MATLAB implementations of the important algorithms. MATLAB code is easy to understand[1], easy to debug, and easy to translate to other languages. The users can even directly use the provided MATLAB code without understanding it.

This paper covers the following topics:

- Chapter 2 briefly reviews some matrix algebra preliminaries. This chapter can be skipped if the reader is familiar with matrix algebra.
- Chapter 3 introduces the techniques for generating a sketch of a large-scale matrix.
- Chapter 4 studies the least squares regression (LSR) problem where $n \gg d$.
- Chapter 5 studies efficient algorithms for computing the k-SVD of arbitrary matrices.

[1] If your are unfamiliar with a MATLAB function, you can simply type "help + functionname" in MATLAB and read the documentation.

- Chapter 6 introduces techniques for sketching symmetric positive semi-definite (SPSD) matrices. The applications includes spectral clustering, kernel methods (e.g. Gaussian process regression and kernel PCA), and second-order optimization (e.g. Newton's method).

Chapter 2

Elementary Matrix Algebra

This chapter defines the matrix notation and goes through the very basics of matrix decompositions. Particularly, the singular value decomposition (SVD), the QR decomposition, and the Moore-Penrose inverse are used throughout this paper.

2.1 Notation

Let $\mathbf{A} = [a_{ij}]$ be a matrix, $\mathbf{a} = [a_i]$ be a column vector, and a be a scalar. The i-th row and j-th column of $\mathbf{A}$ are denoted by $\mathbf{a}_{i:}$ and $\mathbf{a}_{:j}$, respectively. When there is no ambiguity, either column or row can be written as $\mathbf{a}_l$. Let $\mathbf{I}_n$ be the $n \times n$ identity matrix, that is, the diagonal entries are ones and off-diagonal entries are zeros. The column space (the space spanned by the columns) of $\mathbf{A}$ is the set of all possible linear combinations of its column vectors. Let $[n]$ be the set $\{1, 2, \cdots, n\}$. Let $\mathrm{nnz}(\mathbf{A})$ be the number of nonzero entries of $\mathbf{A}$.

The squared vector ℓ_2 norm is defined by

$$\|\mathbf{a}\|_2^2 \;=\; \sum_i a_i^2.$$

The squared matrix Frobenius norm is defined by

$$\|\mathbf{A}\|_F \;=\; \sum_{ij} a_{ij}^2,$$

and the matrix spectral norm is defined by

$$\|\mathbf{A}\|_2 \;=\; \max_{\mathbf{x}\neq\mathbf{0}} \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2}.$$

2.2 Matrix Decompositions

QR decomposition. Let $\mathbf{A}$ be an $m \times n$ matrix with $m \geq n$. The QR decomposition of $\mathbf{A}$ is

$$\mathbf{A} \;=\; \underbrace{\mathbf{Q}_\mathbf{A}}_{m\times n}\underbrace{\mathbf{R}_\mathbf{A}}_{n\times n}.$$

The matrix $\mathbf{Q_A}$ has orthonormal columns, that is, $\mathbf{Q_A}^T\mathbf{Q_A} = \mathbf{I}_n$. The matrix $\mathbf{R_A}$ is upper triangular, that is, for all $i < j$, the (i, j)-th entry of $\mathbf{R_A}$ is zero.

SVD. Let $\mathbf{A}$ be an $m \times n$ matrix and $\rho = \text{rank}(\mathbf{A})$. The condensed singular value decomposition (SVD) of $\mathbf{A}$ is

$$\underbrace{\mathbf{A}}_{m\times n} = \underbrace{\mathbf{U_A}}_{m\times\rho}\underbrace{\mathbf{\Sigma_A}}_{\rho\times\rho}\underbrace{\mathbf{V_A}^T}_{\rho\times n} = \sum_{i=1}^{\rho}\sigma_{\mathbf{A},i}\mathbf{u}_{\mathbf{A},i}\mathbf{v}_{\mathbf{A},i}^T.$$

Here $\sigma_{\mathbf{A},1} \geq \cdots \geq \sigma_{\mathbf{A},\rho} > 0$ are the singular values, $\mathbf{u}_{\mathbf{A},1}, \cdots, \mathbf{u}_{\mathbf{A},\rho} \in \mathbb{R}^m$ are the left singular vectors, and $\mathbf{v}_{\mathbf{A},1}, \cdots, \mathbf{v}_{\mathbf{A},\rho} \in \mathbb{R}^n$ are the right singular vectors. Unless otherwise specified, "SVD" refers to the condensed SVD.

k-SVD. In applications such as the principal component analysis (PCA), latent semantic indexing (LSI), word2vec, spectral clustering, we are only interested in the top k ($\ll m, n$) singular values and singular vectors. The rank k truncated SVD (k-SVD) is denoted by

$$\mathbf{A}_k := \sum_{i=1}^{k}\sigma_{\mathbf{A},i}\mathbf{u}_{\mathbf{A},i}\mathbf{v}_{\mathbf{A},i}^T = \underbrace{\mathbf{U}_{\mathbf{A},k}}_{m\times k}\underbrace{\mathbf{\Sigma}_{\mathbf{A},k}}_{k\times k}\underbrace{\mathbf{V}_{\mathbf{A},k}^T}_{k\times n}.$$

Here $\mathbf{U}_{\mathbf{A},k}$ consists of the first k singular vectors of $\mathbf{U_A}$, and $\mathbf{\Sigma}_{\mathbf{A},k}$ and $\mathbf{V}_{\mathbf{V},k}$ are analogously defined. Among all the $m \times n$ rank k matrices, $\mathbf{A}_k$ is the closest approximation to $\mathbf{A}$ in that

$$\mathbf{A}_k = \underset{\mathbf{X}}{\text{argmin}}\,\|\mathbf{A} - \mathbf{X}\|_F^2 = \underset{\mathbf{X}}{\text{argmin}}\,\|\mathbf{A} - \mathbf{X}\|_2^2, \qquad \text{s.t. rank}(\mathbf{X}) \leq k.$$

Eigenvalue decomposition. The eigenvalue decomposition of an $n \times n$ symmetric matrix $\mathbf{A}$ is defined by

$$\mathbf{A} = \mathbf{U_A}\mathbf{\Lambda_A}\mathbf{U_A}^T = \sum_{i=1}^{n}\lambda_{\mathbf{A},i}\mathbf{u}_{\mathbf{A},i}\mathbf{u}_{\mathbf{A},i}^T.$$

Here $\lambda_{\mathbf{A},1} \geq \cdots \geq \lambda_{\mathbf{A},n}$ are the eigenvalues of $\mathbf{A}$, and $\mathbf{u}_{\mathbf{A},1}, \cdots, \mathbf{u}_{\mathbf{A},n} \in \mathbb{R}^n$ are the corresponding eigenvectors. A symmetric matrix $\mathbf{A}$ is called symmetric positive semidefinite (SPSD) if and only if all the eigenvalues are nonnegative. If $\mathbf{A}$ is SPSD, its SVD and eigenvalue decomposition are identical.

2.3 Matrix (Pseudo) Inverse and Orthogonal Projector

For an $n \times n$ square matrix $\mathbf{A}$, the matrix inverse exists if $\mathbf{A}$ is non-singular ($\text{rank}(\mathbf{A}) = n$). Let $\mathbf{A}^{-1}$ be the inverse of $\mathbf{A}$. Then $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n$.

Only square and full rank matrices have inverse. For the general rectangular matrices or rank deficient matrices, matrix pseudo-inverse is used as a generalization of matrix inverse. The book [1] offers a comprehensive study of the pseudo-inverses.

The Moore-Penrose inverse is the most widely used pseudo-inverse, which is defined by

$$\mathbf{A}^\dagger := \mathbf{V}_\mathbf{A}\mathbf{\Sigma}_\mathbf{A}^{-1}\mathbf{U}_\mathbf{A}^T.$$

Let $\mathbf{A}$ be any $m \times n$ and rank ρ matrix. Then

$$\mathbf{A}\mathbf{A}^\dagger = \mathbf{U}_\mathbf{A}\mathbf{\Sigma}_\mathbf{A}\underbrace{\mathbf{V}_\mathbf{A}^T\mathbf{V}_\mathbf{A}}_{=\mathbf{I}_\rho}\mathbf{\Sigma}_\mathbf{A}^{-1}\mathbf{U}_\mathbf{A}^T = \underbrace{\mathbf{U}_\mathbf{A}}_{m\times\rho}\underbrace{\mathbf{U}_\mathbf{A}^T}_{\rho\times m},$$

which is a orthogonal projector. It is because for any matrix $\mathbf{B}$, the matrix $\mathbf{A}\mathbf{A}^\dagger\mathbf{B} = \mathbf{U}_\mathbf{A}\mathbf{U}_\mathbf{A}^T\mathbf{B}$ is the projection of $\mathbf{B}$ onto the column space of $\mathbf{A}$.

2.4 Time and Memory Costs

The time complexities of the matrix operations are listed in the following.

- Multiplying an $m \times n$ matrix $\mathbf{A}$ by an $n \times p$ matrix $\mathbf{B}$: $\mathcal{O}(mnp)$ float point operations (flops) in general, and $\mathcal{O}(p \cdot \text{nnz}(\mathbf{A}))$ if $\mathbf{A}$ is sparse. Here $\text{nnz}(\mathbf{A})$ is the number of nonzero entries of $\mathbf{A}$.
- QR decomposition, SVD, or Moore-Penrose inverse of an $m \times n$ matrix ($m \geq n$): $\mathcal{O}(mn^2)$ flops.
- k-SVD of an $m \times n$ matrix: $\mathcal{O}(nmk)$ flops (assuming that the spectral gap and the logarithm of error tolerance are constant)
- Matrix inversion or full eigenvalue decomposition of an $n \times n$ matrix: $\mathcal{O}(n^3)$ flops
- k-eigenvalue decomposition of an $n \times n$ matrix: $\mathcal{O}(n^2k)$ flops.

Pass-efficient means that the algorithm goes constant passes through the data. For example, the Frobenius norm of a matrix can be computed pass-efficiently, because each entry is visited only once. In comparison, the spectral norm cannot be computed pass-efficiently, because the algorithm goes at least $\log \frac{1}{\epsilon}$ passes through the matrix, which is not constant. Here ϵ indicates the desired precision.

Memory cost. If an algorithm scans a matrix for constant passes, the matrix can be placed in large volume disks, so the memory cost is not a bottleneck. However, if an algorithm goes through a matrix for many passes (not constant passes), the matrix should be placed in memory, otherwise the swaps between memory and disk would be highly expensive. In this paper, memory cost means the number of entries frequently visited by the algorithm.

Chapter 3

Matrix Sketching

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be the given matrix, $\mathbf{S} \in \mathbb{R}^{n \times s}$ be a sketching matrix, e.g. random projection or column selection matrix, and $\mathbf{C} = \mathbf{AS} \in \mathbb{R}^{m \times s}$ be a sketch of $\mathbf{A}$. The size of $\mathbf{C}$ is much smaller than $\mathbf{A}$, but $\mathbf{C}$ preserves some important properties of $\mathbf{A}$.

3.1 Theoretical Properties

The sketching matrix is useful if it has either or both of the following properties. The two properties are important, and the readers should try to understand them.

Property 3.1 (Subspace Embedding)**.** *For a fixed $m \times n$ ($m \ll n$) matrix $\mathbf{A}$ and all m-dimension vector $\mathbf{y}$, the inequality*

$$\frac{1}{\gamma} \leq \frac{\|\mathbf{y}^T\mathbf{AS}\|_2^2}{\|\mathbf{y}^T\mathbf{A}\|_2^2} \leq \gamma$$

holds with high probability. Here $\mathbf{S} \in \mathbb{R}^{n \times s}$ ($s \ll n$) is a certain sketching matrix.

The subspace embedding property can be intuitively understood in the following way. For all n dimensional vectors $\mathbf{x}$ in the row space of $\mathbf{A}$ (a rank m subspace within $\mathbb{R}^n$),[1] the length of vector $\mathbf{x}$ does not change much after sketching: $\|\mathbf{x}\|_2^2 \approx \|\mathbf{xS}\|_2^2$. This property can be applied to speedup the ℓ_2 regression problems.

Property 3.2 (Low-Rank Approximation)**.** *Let $\mathbf{A}$ be any $m \times n$ matrix and k be any positive integer far smaller than m and n. Let $\mathbf{C} = \mathbf{AS} \in \mathbb{R}^{m \times s}$ where $\mathbf{S} \in \mathbb{R}^{n \times s}$ is a certain sketching matrix and $s \geq k$. The Frobenius norm error bound*[2]

$$\|\mathbf{A} - \mathbf{CC}^\dagger\mathbf{A}\|_F^2 \leq \eta\|\mathbf{A} - \mathbf{A}_k\|_F^2$$

holds with high probability for some $\eta \geq 1$.

[1] Thus there always exists an m dimensional vector y such that $\mathbf{x}$ can be expressed as $\mathbf{x} = \mathbf{y}^T\mathbf{A}$.

[2] Spectral norm bounds should be more interesting. However, spectral norm error is difficult to analyze, and existing spectral norm bounds are "weak" for their factors η are far greater than 1.

The following error bound is stronger and more interesting:

$$\min_{\text{rank}(\mathbf{X})\leq k} \|\mathbf{A}-\mathbf{C}\mathbf{X}\|_F^2 \;\leq\; \eta\|\mathbf{A}-\mathbf{A}_k\|_F^2.$$

It is stronger because $\|\mathbf{A}-\mathbf{C}\mathbf{C}^\dagger\mathbf{A}\|_F^2 \leq \min_{\text{rank}(\mathbf{X})\leq k} \|\mathbf{A}-\mathbf{C}\mathbf{X}\|_F^2$.

Intuitively speaking, the low-rank approximation property means that the columns of $\mathbf{A}_k$ are almost in the column space of $\mathbf{C} = \mathbf{A}\mathbf{P}$. The low-rank approximation property enables us to solve k-SVD more efficiently (for $k \leq s$). Later on we will see that computing the k-SVD of $\mathbf{C}\mathbf{C}^\dagger\mathbf{A}$ is less expensive than the k-SVD of $\mathbf{A}$.

The two properties can be verified by a few lines of MATLAB code. The readers are encouraged to have a try. With a proper sketching method and a relatively large s, both γ and η should be near one.

3.2 Random Projection

The section presents three matrix sketching techniques: Gaussian projection, subsampled randomized Hadamard transform (SRHT), and count sketch. Gaussian projection and SRHT can be combined with count sketch.

3.2.1 Gaussian Projection

The $n \times s$ Gaussian random projection matrix $\mathbf{S}$ is a matrix is formed by $\mathbf{S} = \frac{1}{\sqrt{s}}\mathbf{G}$, where each entry of $\mathbf{G}$ is sampled i.i.d. from $\mathcal{N}(0,1)$. The Gaussian projection is also well knows as the Johnson-Lindenstrauss transform due to the seminal work [15]. Gaussian projection can be implemented in four lines of MATLAB code.

```
function [C] = GaussianProjection(A, s)
n = size(A, 2);
S = randn(n, s) / sqrt(s);
C = A * S;
```

Gaussian projection has the following properties:

- Time cost: $\mathcal{O}(mns)$
- Theoretical guarantees
  1. When $s = \mathcal{O}(m/\epsilon^2)$, the subspace embedding property with $\gamma = 1+\epsilon$ holds with high probability.
  2. When $s = \frac{k}{\epsilon} + 1$, the low-rank approximation property with $\eta = 1+\epsilon$ holds in expectation [3].
- Advantages

1. Easy to implement: four lines of MATLAB code
2. $\mathbf{C}$ is a very high quality sketch of $\mathbf{A}$

- Disadvantages:

1. High time complexity to perform matrix multiplication
2. Sparsity is destroyed: $\mathbf{C}$ is dense even if $\mathbf{A}$ is sparse

3.2.2 Subsampled Randomized Hadamard Transform (SRHT)

The Subsampled Randomized Hadamard Transform (SRHT) matrix is defined by $\mathbf{S} = \frac{1}{\sqrt{sn}}\mathbf{D}\mathbf{H}_n\mathbf{P}$, where

- $\mathbf{D} \in \mathbb{R}^{n\times n}$ is a diagonal matrix with diagonal entries sampled uniformly from $\{+1, -1\}$;
- $\mathbf{H}_n \in \mathbb{R}^{n\times n}$ is defined recursively by

$$\mathbf{H}_n = \begin{bmatrix} \mathbf{H}_{n/2} & \mathbf{H}_{n/2} \\ \mathbf{H}_{n/2} & -\mathbf{H}_{n/2} \end{bmatrix} \qquad \text{and} \qquad \mathbf{H}_2 = \begin{bmatrix} +1 & +1 \\ +1 & -1 \end{bmatrix};$$

  For all $\mathbf{y} \in \mathbb{R}^n$, the matrix vector product $\mathbf{y}^T\mathbf{H}_n$ can be performed in $\mathcal{O}(n \log n)$ time by the fast Walsh–Hadamard transform algorithm in a divide-and-conquer fashion;

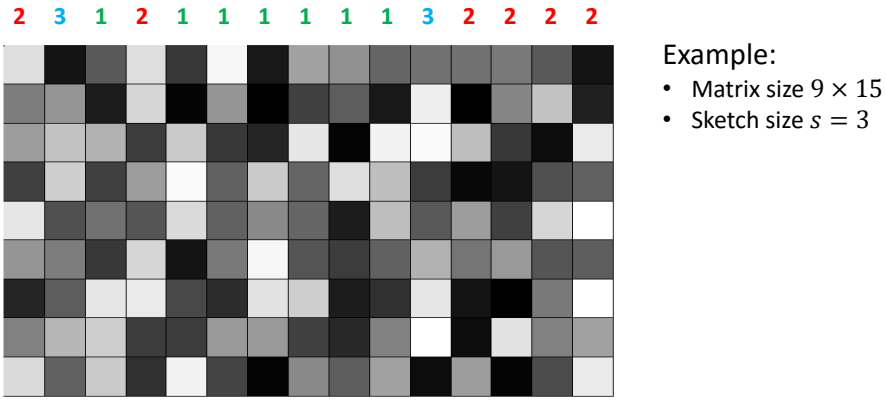
- $\mathbf{P} \in \mathbb{R}^{n\times s}$ samples s from the n columns.

SRHT can be implemented in nine lines of MATLAB code below. Notice that this implementation of SRHT is has $\mathcal{O}(mN \log N)$ ($N \geq n$ is a power of two) time complexity, which is not efficient.

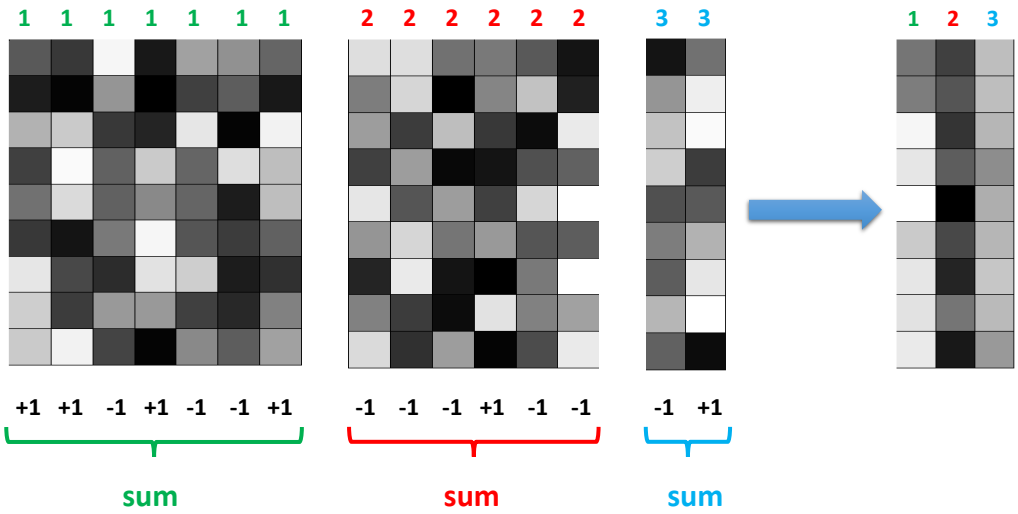
```
function [C] = srht(A, s)
n = size(A, 2);
sgn = randi(2, [1, n]) * 2 - 3; % one half are +1 and the rest are -1
A = bsxfun(@times, A, sgn); % flip the signs of each column w.p. 50%
n = 2^(ceil(log2(n)));
C = (fwht(A', n))'; % fast Walsh-Hadarmard transform
idx = sort(randsample(n, s));
C = C(:, idx); % subsampling
C = C * (n / sqrt(s));
```

The SRHT matrix has the following properties:

- Time complexity: the matrix product $\mathbf{AS}$ can be performed in $\mathcal{O}(mn \log s)$ time, which makes SRHT more efficient than Gaussian projection. (Unfortunately, the MATLAB code above does not have such low time complexity.)
- Theoretical property: when $s = \mathcal{O}(\epsilon^{-2}(m + \log n) \log m)$, SRHT satisfies the subspace embedding property with $\gamma = 1 + \epsilon$ holds with probability 0.99 [34, Theorem 7].



(a) Hash each column with a value uniformly sampled from $[s] = \{1, 2, 3\}$.



(b) Flip the sign of each column with probability 50%, and then sum up columns with the same hash value.

Figure 3.1: Count sketch in the map-reduce fashion.

3.2.3 Count Sketch

Count sketch stems from the data stream literature [4; 26]. It was applied to speedup matrix computation by [6; 21]. We describe in the following the count sketch for matrix data.

There are different ways to implementing count sketch. This paper describe two quite different ways and refer to them as "map-reduce fashion" and "streaming fashion". Of course, the two are equivalent.

- The map-reduce fashion has three steps. First, hash each column with a discrete value uniformly sampled from $[s]$. Second, flip the sign of each column with probability 50%. Third, sum up columns with the same hash value. This procedure is illustrated in

Algorithm 3.1 Count Sketch in the Streaming Fashion.

input: $\mathbf{A} \in \mathbb{R}^{m \times n}$.
Initialize $\mathbf{C}$ to be an $m \times s$ all-zero matrix;
for $i = 1$ to n **do**
    sample l from the set $[s]$ uniformly at random;
    sample g from the set $\{+1, -1\}$ uniformly at random;
    update the l-th column of $\mathbf{C}$ by $\mathbf{c}_{:l} \longleftarrow \mathbf{c}_{:l} + g\mathbf{a}_{:i}$;
end for
return $\mathbf{C} \in \mathbb{R}^{m \times s}$.

Figure 3.1. As its name suggests, this approach naturally fits the map-reduce systems.

- The streaming fashion has two steps. First, initialize $\mathbf{C}$ to be the $m \times s$ all-zero matrix. Second, for each column of $\mathbf{A}$, flip its sign with probability 50%, and add it to a uniformly selected column of $\mathbf{C}$. It is described in Algorithm 3.1 an illustrated in Figure 3.2. It can be implemented in 9 lines of MATLAB code as below. The streaming fashion implementation keeps the sketch $\mathbf{C}$ in memory and scans the data $\mathbf{A}$ in only one pass. If $\mathbf{A}$ does not fit in memory, this approach is better than the map-reduce fashion for it scans the columns sequentially. If $\mathbf{A}$ is sparse matrix, randomly accessing the entries may not be efficient, and thus it is better to accessing the column sequentially.

```
function [C] = CountSketch(A, s) % the streaming fashion
[m, n] = size(A);
sgn = randi(2, [1, n]) * 2 - 3; % one half are +1 and the rest are -1
A = bsxfun(@times, A, sgn); % flip the signs of each column w.p. 50%
ll = randsample(s, n, true); % sample n items from [s] with replacement
C = zeros(m, s); % initialize C
for j = 1: n
    C(:, ll(j)) = C(:, ll(j)) + A(:, j);
end
```

The readers may have noticed that count sketch does not explicitly form the sketching matrix $\mathbf{S}$. In fact, $\mathbf{S}$ is such a matrix that each row has only one nonzero entry. In the example of Figure 3.1, the matrix $\mathbf{S}^T$ can be explicitly expressed as

$$\mathbf{S}^T = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & -1 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Count sketch has the following properties:

- Time cost: $\mathcal{O}(\text{nnz}(\mathbf{A}))$
- Memory cost: $\mathcal{O}(ms)$. When $\mathbf{A}$ does not fit in memory, the algorithm keeps only $\mathbf{C}$ in memory and goes one pass through the columns of $\mathbf{A}$.

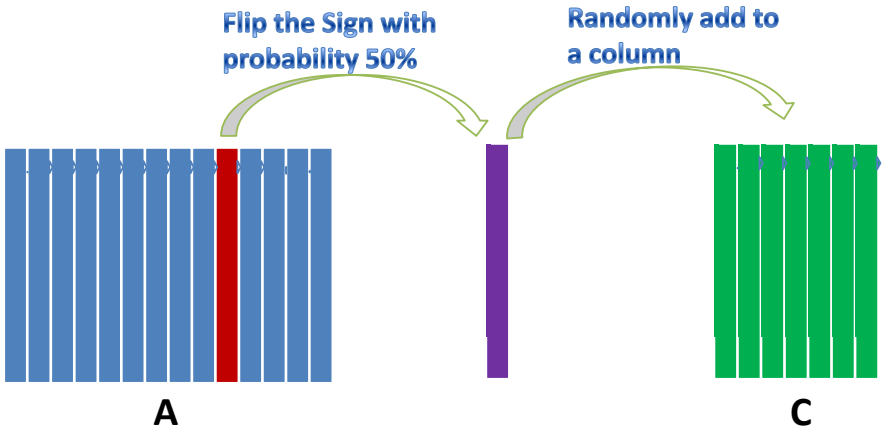


Figure 3.2: Count sketch in the streaming fashion.

- Theoretical guarantees

  1. When $s = \mathcal{O}(m^2/\epsilon^2)$, the subspace embedding property holds with $\gamma = 1+\epsilon$ with high probability.
  2. When $s = \mathcal{O}(k/\epsilon + k^2)$, the low-rank approximation property holds with $\eta = 1+\epsilon$ relative error with high probability.

- Advantage: the count sketch is very efficient, especially when $\mathbf{A}$ is sparse.

- Disadvantage: compared with Gaussian projection, the count sketch requires larger s to attain the same accuracy. One simple improvement is to combine the count sketch with Gaussian projection or SRHT.

3.2.4 GaussianProjection + CountSketch

Let $\mathbf{S}_{sc}$ be $n \times s_{cs}$ count sketch matrix, $\mathbf{S}_{gp}$ be $s_{cs} \times s$ Gaussian projection matrix, and $\mathbf{S} = \mathbf{S}_{cs}\mathbf{S}_{gp} \in \mathbb{R}^{n\times s}$. Then $\mathbf{S}$ satisfies the following properties.

- Time complexity: the matrix product $\mathbf{AS}$ can be computed in

$$\mathcal{O}\Big(\underbrace{\text{nnz}(\mathbf{A})}_{\text{count sketch}} + \underbrace{ms_{cs}s}_{\text{Gaussian projection}} \Big)$$

  time.

- Theoretical properties:

1. When $s_{cs} = \mathcal{O}(m^2/\epsilon^2)$ and $s = \mathcal{O}(m/\epsilon^2)$, the GaussianProjection+CountSketch matrix $\mathbf{S}$ satisfy the subspace embedding property with $\gamma = 1+\epsilon$ holds with high probability.
2. When $s_{cs} = \mathcal{O}(k^2 + k/\epsilon)$ and $s = \mathcal{O}(k/\epsilon)$, the GaussianProjection+CountSketch matrix $\mathbf{S}$ satisfies the low-rank approximation property with $\eta = 1+\epsilon$ [2, Lemma 12].

- Advantages:

1. the size of GaussianProjection+CountSketch is as small as Gaussian projection.
2. the time complexity is much lower than Gaussian projection when $n \gg m$.

3.3 Column Selection

This section presents three column selection techniques: uniform sampling, leverage score sampling, and local landmark selection. Different from random projection, column selection do not have to visit every entry of $\mathbf{A}$, and column selection preserves the sparsity/non-negativity properties of $\mathbf{A}$.

3.3.1 Uniform Sampling

Uniform sampling is the most efficient way to form a sketch. The most important advantage is that uniform sampling forms a sketch without seeing the whole data matrix. When applied to kernel methods, uniform sampling avoids computing every entry of the kernel matrix.

The performance of uniform sampling is data-dependent. When the leverage scores (defined in Section 3.3.2) are uniform, or equivalently, the matrix coherence (namely the greatest leverage score) is small, uniform sampling has good performance. The analysis of uniform sampling can be found in [12; 13].

3.3.2 Leverage Score Sampling

Before studying leverage score sampling, let's first define leverage scores. Let $\mathbf{A}$ be an $m \times n$ matrix, with $\rho = \text{rank}(\mathbf{A}) < n$, and $\mathbf{V} \in \mathbb{R}^{n\times\rho}$ be the right singular vectors. The (column) leverage scores of $\mathbf{A}$ are defined by

$$l_i \ := \ \|\mathbf{v}_{i:}\|_2^2, \qquad \text{for } i = 1, \cdots, n.$$

Leverage score sampling is to select each columns of $\mathbf{A}$ with probability proportional to its leverage scores. (Sometimes each selected column should be scaled by $\sqrt{\frac{\rho}{sl_i}}$.) It can be roughly implemented in 8 lines MATLAB code.

```
1 function [C, idx] = LeverageScoreSampling(A, s)
2 n = size(A, 2);
3 [~, ~, V] = svd(A, 'econ');
4 leveragescores = sum(V.^2, 2);
5 prob = leveragescores / sum(leveragescores);
6 idx = randsample(n, s, true, prob);
7 idx = unique(idx); % eliminate duplicates
8 C = A(:, idx);
```

There are a few things to remark:

- To sample columns according to the leverage scores of $\mathbf{A}_k$ where $k \ll m, n$, Line 3 can be replaced by

```
3 [~, ~, V] = svds(A, k);
```

- Theoretical properties

  1. When $s = \mathcal{O}(m/\epsilon + m \log m)$, the leverage score sampling satisfies the subspace embedding property with $\gamma = 1 + \epsilon$ holds with high probability.
  2. When $s = \mathcal{O}(k/\epsilon + k \log k)$, the leverage score sampling (according to the leverage scores of $\mathbf{A}_k$) satisfies the low-rank approximation property with $\eta = 1 + \epsilon$.

- Computing the leverage scores is as expensive as computing SVD, so leverage score sampling is not a practical way to sketch the matrix $\mathbf{A}$ itself.

- When the leverage scores are near uniform, there is little difference between uniform sampling and leverage score sampling.

3.3.3 Local Landmark Selection

Local landmark selection is a very effective heuristic for finding representative columns. Zhang and Kwok [36] proposed to set $k = s$ and run k-means or k-centroids clustering algorithm to cluster the columns of $\mathbf{A}$ to s class, and use the s centroids as the sketch of $\mathbf{A}$. This heuristic works very well in practice, though it has little theoretical guarantee.

There are several tricks to make the local landmark selection more efficient.

- One can simply solve k-centroids clustering approximately rather than accurately. For example, it is unnecessary to wait for k-centroids clustering to converge; running k-centroids for a few iterations suffices.

- When n is large, one can uniformly sample a subset of the data, e.g. $\max\{0.2n, 20s\}$ data points, and perform local landmark selection on this smaller dataset.

- In supervised learning problems, each datum $\mathbf{a}_i$ is associated with a label y_i. We can partition the data to g groups according to the labels and run k-centroids clustering independently on the data in each group. In this way, $s = gk$ data points are selected as a sketch of $\mathbf{A}$.

Chapter 4

Regression

Let $\mathbf{A}$ be an $n \times d$ ($n \geq d$) matrix whose rows correspond to data and columns correspond to features, and let $\mathbf{b} \in \mathbb{R}^n$ contain the response/label of each datum. The least squares regression (LSR)

$$\min_{\mathbf{x}} \ \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2 \tag{4.1}$$

is a ubiquitous problem in statistics, computer science, economics, etc. When $n \gg d$, LSR can be efficiently solved using randomized algorithms.

4.1 Standard Solutions

The least squares regression (LSR) problem (4.1) has closed form solution

$$\mathbf{x}^\star \ = \ \mathbf{A}^\dagger \mathbf{b}.$$

The Moore-Penrose inverse can be computed by SVD which costs $\mathcal{O}(nd^2)$ time.

LSR can also be solved by numerical algorithms such as the conjugate gradient (CG) algorithm, and machine-precision can be attained in a reasonable number of iterations. Let $\kappa(\mathbf{A}) := \frac{\sigma_1(\mathbf{A})}{\sigma_d(\mathbf{A})}$ be the condition number of $\mathbf{A}$. The convergence of CG depends on $\kappa(\mathbf{A})$:

$$\frac{\|\mathbf{A}(\mathbf{x}^{(t)} - \mathbf{x}^\star)\|_2^2}{\|\mathbf{A}(\mathbf{x}^{(0)} - \mathbf{x}^\star)\|_2^2} \ \leq \ 2\left(\frac{\kappa(\mathbf{A}) - 1}{\kappa(\mathbf{A}) + 1}\right)^t,$$

where $\mathbf{x}^{(t)}$ is the model in the t-th iteration of CG. The per-iteration time cost of CG is $\mathcal{O}(\text{nnz}(\mathbf{A}))$. To attain $\|\mathbf{A}(\mathbf{x}^{(t)} - \mathbf{x}^\star)\|_2^2 \leq \epsilon$, the number of iteration is roughly

$$\left(\log \frac{1}{\epsilon} + \log(\text{InitialError})\right) \frac{\kappa(\mathbf{A}) - 1}{2}.$$

Since the time cost of CG heavily depends on the unknown condition number $\kappa(\mathbf{A})$, CG can be very slow if $\mathbf{A}$ is ill-conditioned.

4.2 Inexact Solution

Any sketching matrix $\mathbf{S} \in \mathbb{R}^{n \times s}$ can be used to solve LSR approximately as long as it satisfies the subspace embedding property. We consider the following LSR problem:

$$\tilde{\mathbf{x}} \;=\; \min_{\mathbf{x}} \; \| \underbrace{(\mathbf{S}^T\mathbf{A})}_{s \times d} \mathbf{x} - \mathbf{S}^T\mathbf{b} \|_2^2, \tag{4.2}$$

which can be solved in $\mathcal{O}(sd^2)$ time.

If $\mathbf{S}$ is a Gaussian projection matrix, SRHT matrix, count sketch, or leverage score sampling matrix, and $s = \text{poly}(d/\epsilon)$ for any error parameter $\epsilon \in (0, 1]$, then

$$\|\mathbf{A}\tilde{\mathbf{x}} - \mathbf{b}\|_2^2 \;\leq\; (1+\epsilon)^2 \min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

is guaranteed.

4.2.1 Implementation

If $\mathbf{S}$ is count sketch matrix, the inexact LSR algorithm can be implemented in 5 lines of MATLAB code. Here CountSketch is a MATLAB function described in Section 3.2.3. The total time cost is $\mathcal{O}(\text{nnz}(\mathbf{A}) + \text{poly}(d/\epsilon))$ and memory cost is $\mathcal{O}(\text{poly}(d/\epsilon))$, which are lower than the cost of exact LSR when $d \ll n$.

```
function [xtilde] = InexactLSR(A, b, s)
d = size(A, 2);
sketch = (CountSketch([A, b]', s))';
Asketch = sketch(:, 1:d); % Asketch = S' * A
bsketch = sketch(:, end); % bsketch = S' * b
xtilde = Asketch \ bsketch;
```

There are a few things to remark:

- The inexact LSR is useful only when $n = \Omega(d/\epsilon + d^2)$.
- The size of sketch s is a polynomial function of ϵ^{-1} rather than logarithm of ϵ^{-1}, thus the algorithm cannot attain high precision.

4.2.2 Theoretical Explanation

By the subspace embedding property, it can be easily shown that $\tilde{\mathbf{x}}$ is a good solution. Let $\mathbf{D} = [\mathbf{A}, \mathbf{b}] \in \mathbb{R}^{n \times (d+1)}$ and $\mathbf{z} = [\mathbf{x}; -1] \in \mathbb{R}^{n+1}$. Then

$$\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{D}\mathbf{z} \qquad \text{and} \qquad \mathbf{S}^T\mathbf{A}\mathbf{x} - \mathbf{S}^T\mathbf{b} = \mathbf{S}^T\mathbf{D}\mathbf{z},$$

and the subspace embedding property indicates $\frac{1}{\eta}\|\mathbf{Dz}\|_2^2 \le \|\mathbf{S}^T\mathbf{Dz}\|_2^2 \le \eta\|\mathbf{Dz}\|_2^2$ for all $\mathbf{z}$. Thus

$$\frac{1}{\eta}\|\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b}\|_2^2 \le \|\mathbf{S}^T(\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b})\|_2^2 \qquad \text{and} \qquad \|\mathbf{S}^T(\mathbf{A}\mathbf{x}^\star-\mathbf{b})\|_2^2 \le \eta\|\mathbf{A}\mathbf{x}^\star-\mathbf{b}\|_2^2$$

The optimality of $\tilde{\mathbf{x}}$ indicates $\|\mathbf{S}^T(\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b})\|_2^2 \le \|\mathbf{S}^T(\mathbf{A}\mathbf{x}^\star-\mathbf{b})\|_2^2$, and thus

$$\begin{aligned}&\frac{1}{\eta}\|\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b}\|_2^2 \le \|\mathbf{S}^T(\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b})\|_2^2 \le \|\mathbf{S}^T(\mathbf{A}\mathbf{x}^\star-\mathbf{b})\|_2^2 \le \eta\|\mathbf{A}\mathbf{x}^\star-\mathbf{b}\|_2^2.\\ &\Rightarrow \|\mathbf{A}\tilde{\mathbf{x}}-\mathbf{b}\|_2^2 \le \eta^2\|\mathbf{A}\mathbf{x}^\star-\mathbf{b}\|_2^2.\end{aligned}$$

Therefore, as long as $\mathbf{S}$ satisfies the subspace embedding property, the approximate solution to LSR is nearly as good as the optimal solution (in terms of objective function value).

4.3 Machine-Precision Solution

Randomized algorithms can also be applied to find machine-precision solution to LSR, and the time complexity is lower than the standard solutions. The state-of-the-art algorithm [18] is based on very similar idea described in this section.

4.3.1 Basic Idea: Preconditioning

We have discussed previously that the time cost of the conjugate gradient (CG) algorithm is roughly

$$\frac{\kappa(\mathbf{A})-1}{2}\Big(\log\frac{1}{\epsilon}+\log(\text{InitialError})\Big)\text{nnz}(\mathbf{A}),$$

which dependents on the condition number of $\mathbf{A}$. To make CG efficient, one can find a $d\times d$ preconditioning matrix $\mathbf{T}$ such that $\kappa(\mathbf{AT})$ is small, solve

$$\mathbf{z}^\star = \operatorname*{argmin}_{\mathbf{z}}\|(\mathbf{AT})\mathbf{z}-\mathbf{b}\|_2^2 \tag{4.3}$$

by CG, and let $\mathbf{x}^\star=\mathbf{Tz}^\star$. In this way, the time cost of CG is roughly

$$\frac{\kappa(\mathbf{AT})-1}{2}\Big(\log\frac{1}{\epsilon}+\log(\text{InitialError})\Big)\text{nnz}(\mathbf{A}).$$

If $\kappa(\mathbf{AT})$ is a small constant, e.g. $\kappa(\mathbf{AT})=2$, then (4.3) can be very efficiently solved by CG.

Now let's consider how to find the preconditioning matrix $\mathbf{T}$. Let $\mathbf{A}=\mathbf{Q_A}\mathbf{R_A}$ be the QR decomposition. Obviously $\mathbf{T}=\mathbf{R_A^{-1}}$ is a perfect preconditioning matrix because $\kappa(\mathbf{AR_A^{-1}})=\kappa(\mathbf{Q_A})=1$. Unfortunately, the preconditioning matrix $\mathbf{T}=\mathbf{R_A^{-1}}$ is not a practical choice because computing the QR decomposition is as expensive as solving LSR.

Woodruff [34] proposed to use sketching to find $\mathbf{R_A}$ approximately in $\mathcal{O}(\text{nnz}(\mathbf{A})+\text{poly}(d))$ time. Let $\mathbf{S}\in\mathbb{R}^{n\times s}$ be a sketching matrix and form $\mathbf{Y}=\mathbf{S}^T\mathbf{A}$. Let $\mathbf{Y}=\mathbf{Q_Y}\mathbf{R_Y}$ be the QR decomposition of $\mathbf{Y}$. Theory shows that the sketch size $s=\mathcal{O}(d^2)$ suffices for $\kappa(\mathbf{AR_Y^{-1}})\le 2$ holding with high probability. Thus $\mathbf{R_Y^{-1}}\in\mathbb{R}^{d\times d}$ is a good preconditioning matrix.

Algorithm 4.1 Machine-Precision Solution to LSR.

1: **input:** $\mathbf{A} \in \mathbb{R}^{n\times d}$, $\mathbf{b} \in \mathbb{R}^n$, and step size θ.
2: Draw a sketching matrix $\mathbf{S} \in \mathbb{R}^{n\times s}$ where $s = \mathcal{O}(d^2)$;
3: Form the sketch $\mathbf{Y} = \mathbf{S}^T\mathbf{A} \in \mathbb{R}^{s\times d}$;
4: Compute the QR decomposition $\mathbf{Y} = \mathbf{Q}_\mathbf{Y}\mathbf{R}_\mathbf{Y}$;
5: Compute the preconditioning matrix $\mathbf{T} = \mathbf{R}_\mathbf{Y}^{-1}$;
6: Compute the initial solution $\mathbf{z}^{(0)} = (\mathbf{S}^T\mathbf{A}\mathbf{T})^\dagger(\mathbf{S}^T\mathbf{b}) = \mathbf{Q}_\mathbf{Y}^T(\mathbf{S}^T\mathbf{b})$;
7: **for** $t = 1, \cdots, \mathcal{O}(\log \epsilon^{-1})$ **do**
8: $\quad \mathbf{r}^{(t)} = \mathbf{b} - \mathbf{A}\mathbf{T}\mathbf{z}^{(t-1)}$; // the residual
9: $\quad \mathbf{z}^{(t)} = \mathbf{z}^{(t-1)} + \theta\mathbf{T}^T\mathbf{A}^T\mathbf{r}^{(t)}$; // gradient descent
10: **end for**
11: **return** $\mathbf{x}^\star = \mathbf{T}\mathbf{z}^{(t)} \in \mathbb{R}^d$.

4.3.2 Algorithm Description

The algorithm is described in Algorithm 4.1. We first form a sketch $\mathbf{Y} = \mathbf{S}^T\mathbf{A} \in \mathbb{R}^{s\times d}$ and compute its QR decomposition $\mathbf{Y} = \mathbf{Q}_\mathbf{Y}\mathbf{R}_\mathbf{Y}$. We can use this QR decomposition to find the initial solution $\mathbf{z}^{(0)}$ and the preconditioning matrix $\mathbf{T} = \mathbf{R}_\mathbf{Y}^{-1}$. If we set $s = \mathcal{O}(d^2)$, the initial solution is only constant times worse than the optimal in terms of objective function value. Theory also ensures that the condition number $\kappa(\mathbf{A}\mathbf{T}) \le 2$. With the good initialization and good condition number, the vanilla gradient descent[1] or CG takes only $\mathcal{O}(\log \epsilon^{-1})$ steps to attain $1+\epsilon$ solution. Notice that Lines 8 and 9 in the algorithm should be cautiously implemented. Do not compute the matrix product $\mathbf{A}\mathbf{T}$ because it would take $\mathcal{O}(\mathrm{nnz}(\mathbf{A})d)$ time!

4.4 Extension: CX-Type Regression

Given any matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, CX decomposition considers decomposing $\mathbf{A}$ into $\mathbf{A} \approx \mathbf{C}\mathbf{X}^\star$, where $\mathbf{C} \in \mathbb{R}^{m\times c}$ is a sketch of $\mathbf{A}$ and $\mathbf{X}^\star \in \mathbb{R}^{c\times n}$ is computed by

$$\mathbf{X}^\star \;=\; \operatorname*{argmin}_{\mathbf{X}} \left\|\mathbf{A} - \mathbf{C}\mathbf{X}\right\|_F^2 \;=\; \mathbf{C}^\dagger\mathbf{A}.$$

It takes $\mathcal{O}(mnc)$ time to compute $\mathbf{X}^\star$. If $c \ll m$, this problem can be solved more efficiently by sketching. Specifically, we can draw a sketching matrix $\mathbf{S} \in \mathbb{R}^{m\times s}$ and compute the approximate solution

$$\tilde{\mathbf{X}} \;=\; \operatorname*{argmin}_{\mathbf{X}} \|\underbrace{\mathbf{S}^T\mathbf{C}}_{s\times c}\underbrace{\mathbf{X}}_{c\times n} - \underbrace{\mathbf{S}^T\mathbf{A}}_{s\times n}\|_F^2 \;=\; (\mathbf{S}^T\mathbf{C})^\dagger(\mathbf{S}^T\mathbf{A})$$

If $\mathbf{S}$ is a count sketch matrix, we set $s = \mathcal{O}(c/\epsilon + c^2)$; if $\mathbf{S}$ samples columns according to the row leverage scores of $\mathbf{C}$, we set $s = \mathcal{O}(c/\epsilon + c\log c)$. It holds with high probability that

$$\left\|\mathbf{A} - \mathbf{C}\tilde{\mathbf{X}}\right\|_F^2 \;\le\; (1+\epsilon)\min_{\mathbf{X}} \left\|\mathbf{A} - \mathbf{C}\mathbf{X}\right\|_F^2.$$

[1] Since $\mathbf{A}\mathbf{T}$ is well conditioned, the vanilla gradient descent and CG has little difference.

4.5 Extension: CUR-Type Regression

A more complicated problem has also been considered in the literature [25; 29; 24]:

$$\mathbf{X}^\star \;=\; \operatorname*{argmin}_{\mathbf{X}} \| \underbrace{\mathbf{C}}_{n\times c} \underbrace{\mathbf{X}}_{c\times r} \underbrace{\mathbf{R}}_{r\times n} - \underbrace{\mathbf{A}}_{m\times n} \|_F^2 \tag{4.4}$$

where $c, r \ll m, n$. The solution is:

$$\mathbf{X}^\star \;=\; \mathbf{C}^\dagger \mathbf{A} \mathbf{R}^\dagger,$$

which cost $\mathcal{O}(mn \cdot \min\{c, r\})$ time. Wang et al. [31] proposed an algorithm to solve (4.4) approximately by

$$\tilde{\mathbf{X}} \;=\; \operatorname*{argmin}_{\mathbf{X}} \|\mathbf{S}_C^T(\mathbf{CXR} - \mathbf{A})\mathbf{S}_R\|_F^2$$

where $\mathbf{S}_C \in \mathbb{R}^{m\times s_c}$ and $\mathbf{S}_R \in \mathbb{R}^{n\times s_r}$ are leverage score sampling matrices. When $s_c = c\sqrt{q/\epsilon}$ and $s_r = r\sqrt{q/\epsilon}$ (where $q = \min\{m, n\}$), it holds with high probability that

$$\|\mathbf{C}\tilde{\mathbf{X}}\mathbf{R} - \mathbf{A}\|_F^2 \;\le\; (1+\epsilon) \min_{\mathbf{X}} \|\mathbf{CXR} - \mathbf{A}\|_F^2.$$

The total time cost is

$$\mathcal{O}(s_c s_r \cdot \min\{c, r\}) \;=\; \mathcal{O}(cr\epsilon^{-1} \cdot \min\{m, n\} \cdot \min\{c, r\})$$

time, which is useful when $\max\{m, n\} \gg c, r$. The algorithm can be implemented in 4 lines of MATLAB code:

```
function [Xtilde] = InexactCurTypeRegression(C, R, A, sc, sr)
[~, idxC] = LeverageScoreSampling(C', sc);
[~, idxR] = LeverageScoreSampling(R, sr);
Xtilde = pinv(C(idxC, :)) * A(idxC, idxR) * pinv(R(:, idxR));
```

Here the function "LeverageScoreSampling" is described in Section 3.3.2. Empirically, setting $s_1 = s_2 = \mathcal{O}(d_1 + d_2)$ suffices for high precision. The experiments in [31] indicates that uniform sampling performs equally well as leverage score sampling.

Chapter 5

Rank k Singular Value Decomposition

This chapter considers the k-SVD of a large scale matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, which may not fit in memory.

5.1 Standard Solutions

The standard solutions to k-SVD include the power iteration algorithm and the Krylov subspace methods. Their time complexities are considered to be $\tilde{\mathcal{O}}(mnk)$, where the $\tilde{\mathcal{O}}$ notation hides parameters such as the spectral gap and logarithm of error tolerance. Here we introduce a simplified version of the block Lanczos method [19][1] which costs time $\mathcal{O}(mnkq)$, where $q = \log \frac{n}{\epsilon}$ is the number of iterations, and the inherent constant depends weakly on the spectral gap. The block Lanczos algorithm is described in Algorithm 5.1 can be implemented in 18 lines of MATLAB code.

```
function [U, S, V] = BlockLanczos(A, k, q)
s = 2 * k; % can be tuned
[m, n] = size(A);
C = A * randn(n, s);
Krylov = zeros(m, s * q);
Krylov(:, 1:s) = C;
for i = 2: q
    C = A' * C;
    C = A * C;
    [C, ~] = qr(C, 0); % optional
    Krylov(:, (i-1)*s+1: i*s) = C;
end
[Q, ~] = qr(Krylov, 0);
[Ubar, S, V] = svd(Q' * A, 'econ');
Ubar = Ubar(:, 1:k);
S = S(1:k, 1:k);
```

[1] We introduce this algorithm because it is easy to understand. However, as q grows, columns of the Krylov matrix gets increasingly linearly dependent, which sometimes leads to instability. Thus there are many numerical treatments to strengthen stability (see the numerically stable algorithms in [23]).

Algorithm 5.1 k-SVD by the Block Lanczos Algorithm.

Input: an $m \times n$ matrix $\mathbf{A}$ and the target rank k.
Set $s = k + \mathcal{O}(1)$ be the over-sampling parameter;
Set $q = \mathcal{O}(\log \frac{n}{\epsilon})$ be the number of iteration;
Draw a $n \times s$ sketching matrix $\mathbf{S}$;
$\mathbf{C} = \mathbf{AS}$;
Set $\mathbf{K} = \big[\mathbf{C},\ (\mathbf{AA}^T)\mathbf{C},\ (\mathbf{AA}^T)^2\mathbf{C},\ \cdots,\ (\mathbf{AA}^T)^{q-1}\mathbf{C}\big]$;
QR decomposition: $[\underbrace{\mathbf{Q}_\mathbf{C}}_{m\times sq}, \mathbf{R}_\mathbf{C}] = qr(\underbrace{\mathbf{K}}_{m\times sq})$;
SVD: $[\underbrace{\bar{\mathbf{U}}}_{sq\times sq}, \underbrace{\mathbf{\Sigma}}_{sq\times sq}, \underbrace{\mathbf{V}}_{n\times sq}] = svd(\underbrace{\mathbf{Q}_\mathbf{C}^T\mathbf{A}}_{s\times n})$;
Retain the top k components of $\bar{\mathbf{U}}$, $\mathbf{\Sigma}$, and $\mathbf{V}$ to form $sq \times k$, $k \times k$, $n \times k$ matrices;
$\mathbf{U} = \mathbf{Q}\bar{\mathbf{U}} \in \mathbb{R}^{m\times k}$;
return $\mathbf{U\Sigma V}^T \approx \mathbf{A}_k$.

```
V = V(:, 1:k);
U = Q * Ubar;
```

Although the block Lanczos algorithm can attain machine precision, it inevitably goes many passes through $\mathbf{A}$, and it is thus slow when $\mathbf{A}$ does not fit in memory.

Facing large-scale data, we must trade off between precision and computational costs. We are particularly interested in approximate algorithm that satisfies:

1. The algorithm goes constant passes through $\mathbf{A}$. Then $\mathbf{A}$ can be stored in large volume disks, and there are only constant swaps between disk and memory.

2. The algorithm only keeps a small-scale sketch of $\mathbf{A}$ in memory.

3. The time cost is $\mathcal{O}(mnk)$ or lower.

5.2 Prototype Randomized k-SVD Algorithm

This section describes a randomized algorithm that computes the k-SVD of $\mathbf{A}$ up to $1+\epsilon$ Frobenius norm relative error. The algorithm is proposed by [14], and it is described in Algorithm 5.2.

5.2.1 Theoretical Explanation

If $\mathbf{C} = \mathbf{AS} \in \mathbb{R}^{m\times s}$ is a good sketch of $\mathbf{A}$, the column space of $\mathbf{C}$ should roughly contain the columns of $\mathbf{A}_k$—this is the low-rank approximation property. If $\mathbf{S} \in \mathbb{R}^{n\times s}$ is Gaussian projection matrix or count sketch and $s = \mathcal{O}(k/\epsilon)$, then the low-rank approximation property

$$\min_{\text{rank}(\mathbf{Z})\le k} \|\mathbf{CZ} - \mathbf{A}\|_F^2 \ \le\ (1+\epsilon)\|\mathbf{A} - \mathbf{A}_k\|_F^2 \tag{5.1}$$

Algorithm 5.2 Prototype Randomized k-SVD Algorithm.

1: **Input:** an $m \times n$ matrix $\mathbf{A}$ and the target rank k.
2: Draw a $n \times s$ sketching matrix $\mathbf{S}$ where $s = \mathcal{O}(\frac{k}{\epsilon})$;
3: $\mathbf{C} = \mathbf{AS}$;
4: QR decomposition: $[\underbrace{\mathbf{Q_C}}_{m\times s}, \mathbf{R_C}] = qr(\underbrace{\mathbf{C}}_{m\times s})$;
5: k-SVD: $[\underbrace{\bar{\mathbf{U}}}_{s\times k}, \underbrace{\tilde{\mathbf{\Sigma}}}_{k\times k}, \underbrace{\tilde{\mathbf{V}}}_{n\times k}] = svds(\underbrace{\mathbf{Q_C^T A}}_{s\times n}, k)$;
6: $\tilde{\mathbf{U}} = \mathbf{Q_C}\bar{\mathbf{U}} \in \mathbb{R}^{m\times k}$;
7: **return** $\tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^T \approx \mathbf{A}_k$.

holds in expectation.

5.2.2 Algorithm Derivation

Let $\mathbf{Q_C}$ be any orthonormal bases of $\mathbf{C}$. Since the column space of $\mathbf{C}$ is the same to the column space of $\mathbf{Q_C}$, the minimization problem in (5.1) can be equivalently converted to

$$\mathbf{X}^\star = \underset{\text{rank}(\mathbf{X})\le k}{\text{argmin}} \| \underbrace{\mathbf{Q_C}}_{m\times s}\underbrace{\mathbf{X}}_{s\times n} - \underbrace{\mathbf{A}}_{m\times n} \|_F^2 = (\mathbf{Q_C^T A})_k. \tag{5.2}$$

Here the second equality is a well known fact. The matrix $\mathbf{A}_k$ is well approximated by $\tilde{\mathbf{A}}_k := \mathbf{Q_C X}^\star$, so we need only to find the k-SVD of $\tilde{\mathbf{A}}_k$:

$$\tilde{\mathbf{A}}_k := \underbrace{\mathbf{Q_C}}_{m\times s}\underbrace{\mathbf{X}^\star}_{s\times n} = \mathbf{Q_C}\underbrace{(\mathbf{Q_C^T A})_k}_{:=\bar{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^T} = \underbrace{\mathbf{Q_C}\bar{\mathbf{U}}}_{:=\tilde{\mathbf{U}}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^T = \underbrace{\tilde{\mathbf{U}}}_{m\times k}\underbrace{\tilde{\mathbf{\Sigma}}}_{k\times k}\underbrace{\tilde{\mathbf{V}}^T}_{k\times n}.$$

It is easy to check that $\tilde{\mathbf{U}}$ and $\tilde{\mathbf{V}}$ have orthonormal columns and $\tilde{\mathbf{\Sigma}}$ is a diagonal matrix. Notice that the accuracy of the randomized k-SVD depends only on the quality of the sketch matrix $\mathbf{C}$.

5.2.3 Implementation

The algorithm is described in Algorithm 5.2 and can be implemented in 5 lines of MATLAB code. Here $s = \mathcal{O}(\frac{k}{\epsilon})$ is the size of the sketch.

```
1 function [Utilde, Stilde, Vtilde] = ksvdPrototype(A, k, s)
2 C = CountSketch(A, s);
3 [Q, R] = qr(C, 0);
4 [Ubar, Stilde, Vtilde] = svds(Q' * A, k);
5 Utilde = Q * Ubar;
```

Empirically, using "svd($Q' * A$, 'econ')" followed by discarding the $k+1$ to s components should be faster than the "svds" function in Line 4.

The algorithm has the following properties:

1. The algorithm goes 2 passes through $\mathbf{A}$;
2. The algorithm only keeps an $m \times \mathcal{O}(\frac{k}{\epsilon})$ sketch $\mathbf{C}$ in memory;
3. The time cost is $\mathcal{O}(\text{nnz}(\mathbf{A})k/\epsilon)$.

5.3 Faster Randomized k-SVD

The prototype algorithm spends most of its time on solving (5.2); if (5.2) can be solved more efficiently, the randomized k-SVD can be even faster. The readers may have noticed that (5.2) is the least squares regression (LSR) problem discussed in Section 4.4. Yes, we can solve (5.2) efficiently by the inexact LSR algorithm presented in the previous section.

5.3.1 Theoretical Explanation

Now we draw a $m \times p$ GaussianProjection+CountSketch matrix $\mathbf{P}$ and solve this problem:

$$\tilde{\mathbf{X}} = \underset{\text{rank}(\mathbf{X}) \le k}{\text{argmin}} \| \underbrace{\mathbf{P}^T \mathbf{Q}_\mathbf{C}}_{p \times s} \underbrace{\mathbf{X}}_{s \times n} - \underbrace{\mathbf{P}^T \mathbf{A}}_{p \times n} \|_F^2. \tag{5.3}$$

To understand this trick, the readers can retrospect the extension of LSR in Section 4.4. Let

$$\mathbf{P} = \underbrace{\mathbf{P}_{cs}}_{m \times p_{cs}} \underbrace{\mathbf{P}_{srht}}_{p_{cs} \times p}$$

where $p_{cs} = \mathcal{O}(k/\epsilon + k^2)$ and $p = \mathcal{O}(k/\epsilon)$. The subspace embedding property of RSHT+CountSketch [6, Theorem 46] implies that

$$\begin{aligned}
&(1+\epsilon)^{-1} \|\mathbf{Q}_\mathbf{C} \tilde{\mathbf{X}} - \mathbf{A}\|_F^2 \le \|\mathbf{P}^T(\mathbf{Q}_\mathbf{C} \tilde{\mathbf{X}} - \mathbf{A})\|_F^2 \le \|\mathbf{P}^T(\mathbf{Q}_\mathbf{C} \mathbf{X}^\star - \mathbf{A})\|_F^2 \le (1+\epsilon) \|\mathbf{Q}_\mathbf{C} \mathbf{X}^\star - \mathbf{A}\|_F^2, \\
&\Rightarrow \|\mathbf{Q}_\mathbf{C} \tilde{\mathbf{X}} - \mathbf{A}\|_F^2 \le (1+\epsilon)^2 \|\mathbf{Q}_\mathbf{C} \mathbf{X}^\star - \mathbf{A}\|_F^2 \le (1+\epsilon)^3 \|\mathbf{A} - \mathbf{A}_k\|_F^2.
\end{aligned}$$

Here the second inequality follows from the optimality of $\tilde{\mathbf{X}}$, and the last inequality follows from the low-rank approximation property of the sketch $\mathbf{C} = \mathbf{AS}$. Thus, by solving (5.3) we get k-SVD up to $1 + \mathcal{O}(\epsilon)$ Frobenius norm relative error.

5.3.2 Algorithm Derivation

The faster randomized k-SVD is described in Algorithm 5.3 and derived in the following. The algorithm solves

$$\tilde{\mathbf{X}} = \underset{\text{rank}(\mathbf{X}) \le k}{\text{argmin}} \| \underbrace{\mathbf{P}^T \mathbf{C}}_{p \times s} \underbrace{\mathbf{X}}_{s \times n} - \underbrace{\mathbf{P}^T \mathbf{A}}_{p \times n} \|_F^2 \tag{5.4}$$

to obtain the rank k matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{c \times n}$, and approximates $\mathbf{A}_k$ by

$$\mathbf{A}_k \approx \mathbf{C} \tilde{\mathbf{X}}.$$

Algorithm 5.3 Faster Randomized k-SVD Algorithm.

Input: an $m \times n$ matrix $\mathbf{A}$ and the target rank k.
Set the parameters as $s = \tilde{\mathcal{O}}(\frac{k}{\epsilon})$, $p_{cs} = s^2 \log^6 \frac{s}{\epsilon} + \frac{s}{\epsilon}$, and $p = \frac{s}{\epsilon} \log \frac{s}{\epsilon}$;
Draw a $n \times s$ count sketch matrix $\mathbf{S}$ and perform sketching: $\mathbf{C} = \mathbf{AS}$;
Draw an $m \times p_{cs}$ count sketch matrix $\mathbf{P}_{cs}$ and an $p_{cs} \times p$ matrix $\mathbf{P}_{srht}$;
Perform Sketching: $\mathbf{D} = \mathbf{P}_{srht}^T \mathbf{P}_{cs}^T \mathbf{C} \in \mathbb{R}^{p \times s}$ and $\mathbf{L} = \mathbf{P}_{srht}^T \mathbf{P}_{cs}^T \mathbf{A} \in \mathbb{R}^{p \times n}$;
QR decomposition: $[\underbrace{\mathbf{Q_D}}_{p\times s}, \underbrace{\mathbf{R_D}}_{s\times s}] = qr(\underbrace{\mathbf{D}}_{p\times s})$;
k-SVD: $[\underbrace{\bar{\mathbf{U}}}_{s\times k}, \underbrace{\bar{\mathbf{\Sigma}}}_{k\times k}, \underbrace{\bar{\mathbf{V}}}_{n\times k}] = svds(\underbrace{\mathbf{Q_D^T L}}_{s\times n}, k)$;
SVD: $[\underbrace{\tilde{\mathbf{U}}}_{n\times k}, \underbrace{\tilde{\mathbf{\Sigma}}}_{k\times k}, \underbrace{\hat{\mathbf{V}}}_{k\times k}] = svd(\underbrace{\mathbf{C}\mathbf{R}_\mathbf{D}^\dagger \bar{\mathbf{U}}\bar{\mathbf{\Sigma}}}_{s\times k})$;
$\tilde{\mathbf{V}} = \underbrace{\bar{\mathbf{V}}}_{n\times k} \underbrace{\hat{\mathbf{V}}}_{k\times k}$;
return $\tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^T \approx \mathbf{A}_k$.

Define $\mathbf{D} = \mathbf{P}^T\mathbf{C}$, $\mathbf{L} = \mathbf{P}^T\mathbf{A}$, and let $\mathbf{Q_D R_D} = \mathbf{D}$ be the QR decomposition. Then (5.4) becomes

$$\tilde{\mathbf{X}} = \underset{\text{rank}(\mathbf{X})\le k}{\text{argmin}} \| \underbrace{\mathbf{D}}_{p\times s} \underbrace{\mathbf{X}}_{s\times n} - \underbrace{\mathbf{L}}_{p\times n} \|_F^2 = \underbrace{\mathbf{R}_\mathbf{D}^\dagger}_{s\times s} \underbrace{(\mathbf{Q}_\mathbf{D}^T\mathbf{L})_k}_{s\times n}.$$

Based on the defined notation, we decompose $\mathbf{A}_k \approx \mathbf{C}\tilde{\mathbf{X}}$ by

$$\mathbf{A}_k \approx \mathbf{C}\tilde{\mathbf{X}} = \mathbf{C}\mathbf{R}_\mathbf{D}^\dagger \underbrace{(\mathbf{Q}_\mathbf{D}^T\mathbf{L})_k}_{:=\bar{\mathbf{U}}\bar{\mathbf{\Sigma}}\bar{\mathbf{V}}^T} = \underbrace{\mathbf{C}\mathbf{R}_\mathbf{D}^\dagger \bar{\mathbf{U}}\bar{\mathbf{\Sigma}}}_{:=\tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\hat{\mathbf{V}}^T} \bar{\mathbf{V}}^T = \tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}} \underbrace{\hat{\mathbf{V}}^T\bar{\mathbf{V}}^T}_{:=\tilde{\mathbf{V}}^T} = \underbrace{\tilde{\mathbf{U}}}_{m\times k} \underbrace{\tilde{\mathbf{\Sigma}}}_{k\times k} \underbrace{\tilde{\mathbf{V}}^T}_{k\times n}.$$

5.3.3 Implementation

The faster randomized k-SVD is described in Algorithm 5.3 and implemented in 18 lines of MATLAB code.

```
function [Utilde, Stilde, Vtilde] = ksvdFaster(A, k, s, p1, p2)
n = size(A, 2);
C = CountSketch(A, s);
A = [A, C];
A = A';
sketch = CountSketch(A, p1);
clear A % A (m-by-n) will not be used
sketch = GaussianProjection(sketch, p2);
sketch = sketch';
L = sketch(:, 1:n);
D = sketch(:, n+1:end);
clear sketch % sketch (p2-by-(n+c)) will not be used
[QD, RD] = qr(D, 0);
```

```
[Ubar, Sbar, Vbar] = svds(QD' * L, k);
clear L % L (p2-by-n) will not be used
C = C * (pinv(RD) * (Ubar * Sbar));
[Utilde, Stilde, Vhat] = svd(C, 'econ');
Vtilde = Vbar * Vhat;
```

There are a few things to remark:

1. The algorithm goes only two passes through $\mathbf{A}$.
2. The algorithm costs time $\mathcal{O}\big(\mathrm{nnz}(\mathbf{A}) + (m+n)\mathrm{poly}(k/\epsilon)\big)$.
3. The parameters should be set as $k < s < p2 < p1 \ll m, n$.
4. Line 8 can be removed or replaced by other sketching methods.
5. "A", "sketch", and "L" are the most memory expensive variables in the program, but fortunately, they are swept only one or two passes. If "A", "sketch", and "L" do not fit in memory, they should be stored in disk and loaded to memory block-by-block to perform computations.
6. Unless both m and n are large enough, this algorithm may be slower than the prototype algorithm.

Chapter 6

SPSD Matrix Sketching

This chapter considers SPSD matrix $\mathbf{K} \in \mathbb{R}^{n \times n}$, which can be a kernel matrix, a social network graph, a Hessian matrix, or a Fisher information matrix. Our objective is to find a low-rank decomposition $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$. (Notice that $\mathbf{L}\mathbf{L}^T$ is always SPSD, no matter what $\mathbf{L}$ is.) If $\mathbf{K}$ is symmetric but not SPSD, it can be approximated by $\mathbf{K} \approx \mathbf{C}\mathbf{Z}\mathbf{C}^T$ where $\mathbf{Z}$ is symmetric but not necessarily SPSD.

6.1 Motivations

This section provides three motivation examples to show why we seek to sketch $\mathbf{K}$ by $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$ or $\mathbf{K} \approx \mathbf{C}\mathbf{Z}\mathbf{C}^T$.

6.1.1 Forming a Kernel Matrix

In the kernel approximation problems, we are given

- an $n \times d$ matrix $\mathbf{X}$, whose rows are data points $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$,
- a kernel function, e.g. the Gaussian RBF kernel function defined by

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\Big(-\frac{1}{2\sigma^{-2}} \|\mathbf{x}_i - \mathbf{x}_j\|_2^2 \Big)$$

  where $\sigma > 0$ is the kernel width parameter.

The RBF kernel matrix can be computed by the following MATLAB code:

```
function [K] = rbf(X1, X2, sigma)
K = X1 * X2';
X1_row_sq = sum(X1.^2, 2) / 2;
X2_row_sq = sum(X2.^2, 2) / 2;
K = bsxfun(@minus, K, X1_row_sq);
K = bsxfun(@minus, K, X2_row_sq');
K = K / (sigma^2);
K = exp(K);
```

If $\mathbf{X}_1$ and $\mathbf{X}_2$ are respectively $n_1 \times d$ and $n_2 \times d$ matrices, then the output of "rbf" is an $n_1 \times n_2$ matrix.

Kernel methods requires forming the $n \times n$ kernel matrix $\mathbf{K}$ whose the (i,j)-th entry is $\kappa(\mathbf{x}_i, \mathbf{x}_j)$. The RBF kernel matrix can be computed by the MATLAB function

```
K = rbf(X, X, sigma)
```

in $\mathcal{O}(n^2 d)$ time.

In presence of millions of data points, it is prohibitive to form such a kernel matrix. Fortunately, a sketch of $\mathbf{K}$ can be obtained very efficiently. Let $\mathbf{S} \in \mathbb{R}^{n \times s}$ be a uniform column selection matrix[1] described in Section 3.3, then $\mathbf{C} = \mathbf{KS}$ can be obtained in $\mathcal{O}(nsd)$ time by the following MATLAB code.

```
function [C] = rbfSketch(X, sigma, s)
n = size(X, 1);
idx = sort(randsample(n, s));
C = rbf(X, X(idx, :), sigma);
```

6.1.2 Matrix Inversion

Let $\mathbf{K}$ be an $n \times n$ kernel matrix, $\mathbf{y}$ be an n dimensional vector, and α be a positive constant. Kernel methods such as the Gaussian process regression (or the equivalent the kernel ridge regression) and the least squares SVM require solving

$$(\mathbf{K} + \alpha \mathbf{I}_n)\mathbf{w} \;=\; \mathbf{y}$$

to obtain $\mathbf{w} \in \mathbb{R}^n$. The exact solution costs $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ memory.

If we have a rank l approximation $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$, then $\mathbf{w}$ can be approximately obtained in $\mathcal{O}(nl^2)$ time and $\mathcal{O}(nl)$ memory. Here we need to apply the Sherman-Morrison-Woodbury matrix identity

$$(\mathbf{A} + \mathbf{BCD})^{-1} \;=\; \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{C}^{-1} + \mathbf{D}\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{D}\mathbf{A}^{-1}.$$

We expand $(\mathbf{L}\mathbf{L}^T + \alpha \mathbf{I}_n)^{-1}$ by the above identity and obtain

$$(\mathbf{L}\mathbf{L}^T + \alpha \mathbf{I}_n)^{-1} \;=\; \alpha^{-1}\mathbf{I}_n - \alpha^{-1}\mathbf{L}(\underbrace{\alpha \mathbf{I}_l + \mathbf{L}^T\mathbf{L}}_{l \times l})^{-1}\mathbf{L}^T,$$

and thus

$$\mathbf{w} \;=\; (\mathbf{K} + \alpha \mathbf{I}_n)^{-1}\mathbf{y} \;\approx\; \alpha^{-1}\mathbf{y} - \alpha^{-1}\mathbf{L}(\alpha \mathbf{I}_l + \mathbf{L}^T\mathbf{L})^{-1}\mathbf{L}^T\mathbf{y}.$$

The matrix inversion problem not only appears in the kernel methods, but also in the second order optimization problems. Newton's method and the so-called natural gradient

[1] The local landmark selection is sometimes a better choice. Do not use random projections, because they inevitably visit every entry of $\mathbf{K}$.

method require computing $\mathbf{H}^{-1}\mathbf{g}$, where $\mathbf{g}$ is the gradient and $\mathbf{H}$ is the Hessian matrix or the Fisher information matrix. Since low-rank matrices are not invertible, the naive low-rank approximation $\mathbf{H} \approx \mathbf{C}\mathbf{Z}\mathbf{C}^T$ does not work. To make matrix inversion possible, one can use the spectral shifting trick of [30]: fix a small constant $\alpha > 0$, form the low-rank approximation $\mathbf{H} - \alpha\mathbf{I}_n \approx \mathbf{C}\mathbf{Z}\mathbf{C}^T$, and compute $\mathbf{H}^{-1}\mathbf{g} \approx (\mathbf{C}\mathbf{Z}\mathbf{C}^T + \alpha\mathbf{I}_n)^{-1}\mathbf{g}$. Besides the low-rank approximation approach, one can approximate $\mathbf{H}$ by a block diagonal matrix or even its diagonal, because it is easy to invert a diagonal matrix or a block diagonal matrix.

6.1.3 Eigenvalue Decomposition

With the low-rank decomposition $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$ at hand, we first approximately decompose $\mathbf{K}$ by

$$\mathbf{K} \approx \mathbf{L}\mathbf{L}^T = (\mathbf{U}_\mathbf{L}\mathbf{\Sigma}_\mathbf{L}\mathbf{V}_\mathbf{L}^T)(\mathbf{U}_\mathbf{L}\mathbf{\Sigma}_\mathbf{L}\mathbf{V}_\mathbf{L}^T)^T = \mathbf{U}_\mathbf{L}\mathbf{\Sigma}_\mathbf{L}^2\mathbf{U}_\mathbf{L}^T,$$

and then discard the $k+1$ to l components in $\mathbf{U}_\mathbf{L}$ and $\mathbf{\Sigma}_\mathbf{L}$. Here $\mathbf{L} = \mathbf{U}_\mathbf{L}\mathbf{\Sigma}_\mathbf{L}\mathbf{V}_\mathbf{L}^T$ is the SVD of $\mathbf{L}$, which can be obtained in $\mathcal{O}(nl^2)$ time and $\mathcal{O}(nl)$ memory. In this way, the rank k ($k \le \text{rank}(\mathbf{L})$) eigenvalue decomposition is approximately computed.

6.2 Prototype Algorithm

From now on, we will consider how to find the low-rank approximation $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$. As usual, the simplest approach is to form a sketch $\mathbf{C} = \mathbf{K}\mathbf{S} \in \mathbb{R}^{n\times s}$ and solve

$$\mathbf{X}^\star = \min_\mathbf{X} \|\mathbf{K} - \mathbf{C}\mathbf{X}\mathbf{C}^T\|_F^2 = \mathbf{C}^\dagger\mathbf{K}(\mathbf{C}^\dagger)^T \quad \text{or} \quad \mathbf{Z}^\star = \min_\mathbf{Z} \|\mathbf{K} - \mathbf{Q}_\mathbf{C}\mathbf{Z}\mathbf{Q}_\mathbf{C}\|_F^2 = \mathbf{Q}_\mathbf{C}^T\mathbf{K}\mathbf{Q}_\mathbf{C}, \tag{6.1}$$

where $\mathbf{Q}_\mathbf{C}$ is the orthonormal bases of $\mathbf{C}$ computed by SVD or QR decomposition. It is obvious that $\mathbf{C}\mathbf{X}^\star\mathbf{C} = \mathbf{Q}_\mathbf{C}\mathbf{Z}^\star\mathbf{Q}_\mathbf{C}^T$. In this way, a rank c approximation to $\mathbf{K}$ is obtained. This approach is first studied by [14]. Wang *et al.* [30] showed that if $\mathbf{C}$ contains $s = \mathcal{O}(k/\epsilon)$ columns of $\mathbf{K}$ chosen by adaptive sampling, the error bound

$$\mathbb{E}\|\mathbf{K} - \mathbf{Q}_\mathbf{C}\mathbf{Z}^\star\mathbf{Q}_\mathbf{C}^T\|_F^2 \le (1+\epsilon)\|\mathbf{K} - \mathbf{K}_k\|_F^2$$

is guaranteed. Other sketching methods can also be applied, although currently they do not have $1+\epsilon$ error bound. In the following we implement the prototype algorithm (with the count sketch) in 5 lines of MATLAB code. Since the algorithm goes only two passes through $\mathbf{K}$, when $\mathbf{K}$ does not fit in memory, we can store $\mathbf{K}$ in the disk and keep one block of $\mathbf{K}$ in memory at a time. In this way, $\mathcal{O}(ns)$ memory is enough.

```
function [QC, Z] = spsdPrototype(K, s)
n = size(K, 2);
C = CountSketch(K, s);
[QC, ~] = qr(C, 0);
Z = QC' * K * QC;
```

Algorithm 6.1 Faster SPSD Matrix Sketching.

1: **Input:** an $n \times n$ matrix $\mathbf{K}$ and integers s and p ($s \le p \ll n$).
2: Draw a column selection matrix $\mathbf{S} \in \mathbb{R}^{n \times s}$;
3: Perform sketching: $\mathbf{C} = \mathbf{AS}$;
4: QR decomposition: $[\mathbf{Q_C}, \mathbf{R_C}] = qr(\mathbf{C})$;
5: Draw a column selection matrix $\mathbf{P} \in \mathbb{R}^{n \times p}$;
6: Compute $\tilde{\mathbf{Z}} = (\mathbf{P}^T\mathbf{Q_C})^\dagger(\mathbf{P}^T\mathbf{KP})(\mathbf{Q_C}^T\mathbf{P})^\dagger$;
7: **return** $\mathbf{Q_C}\tilde{\mathbf{Z}}\mathbf{Q_C}^T \approx \mathbf{A}$.

Despite its simplicity, the algorithm has several drawbacks.

- The time cost of this algorithm is $\mathcal{O}(ns^2 + \mathrm{nnz}(\mathbf{K})s)$, which can be quadratic in n.
- The algorithm must visit every entry of $\mathbf{K}$, which can be a serious drawback when applied to kernel methods. It is because computing the kernel matrix $\mathbf{K}$ costs $\mathcal{O}(n^2 d)$ time, where d is the dimension of the data points.

Therefore, we are interested in computing a low-rank approximation in linear time (w.r.t. n) and avoiding visiting every entry of $\mathbf{K}$.

6.3 Faster SPSD Matrix Sketching

The readers may have noticed that (6.1) is the problem studied in Section 4.5. We can thus draw a column selection matrix $\mathbf{P} \in \mathbb{R}^{n \times p}$ and approximately solve (6.1) by

$$\tilde{\mathbf{Z}} \;=\; \min_{\mathbf{Z}} \|\mathbf{P}^T(\mathbf{K} - \mathbf{Q_C}\mathbf{Z}\mathbf{Q_C})\mathbf{P}\|_F^2 \;=\; \underbrace{(\mathbf{P}^T\mathbf{Q_C})^\dagger}_{s\times p}\underbrace{(\mathbf{P}^T\mathbf{KP})}_{p\times p}\underbrace{(\mathbf{Q_C}^T\mathbf{P})^\dagger}_{p\times s}. \tag{6.2}$$

Then we can approximate $\mathbf{K}$ by $\mathbf{Q_C}\tilde{\mathbf{Z}}\mathbf{Q_C}^T$. We describe the faster SPSD matrix sketching in Algorithm 6.1.

There are a few things to remark.

- Since we are trying to avoid computing every entry of $\mathbf{K}$, we should use uniform sampling or local landmark selection to form $\mathbf{C} = \mathbf{KS}$.
- Let $\mathbf{P} \in \mathbb{R}^{n \times p}$ be a leverage score sampling matrix according to the columns of $\mathbf{C}^T$. That is, it samples the i-th column with probability proportional to q_i, where q_i is the squared ℓ_2 norm of the i-th row of $\mathbf{Q_C}$ (for $i = 1$ to n). When $p = \mathcal{O}(\sqrt{n}s\epsilon^{-1/2})$, the following error bounds holds with high probability [31]

$$\|\mathbf{K} - \mathbf{Q_C}\tilde{\mathbf{Z}}\mathbf{Q_C}^T\|_F^2 \;\le\; (1+\epsilon)\min_{\mathbf{Z}} \|\mathbf{K} - \mathbf{Q_C}\mathbf{Z}\mathbf{Q_C}^T\|_F^2.$$

- Let $\mathbf{S}$ be a uniform sampling matrix and $\mathbf{P}$ be a leverage score sampling matrix. The algorithm visits only $ns + p^2 = \mathcal{O}(n)$ entries of $\mathbf{K}$. The overall time and memory costs are linear in n.

- Assume $\mathbf{S}$ is a column selection matrix. Let the sketch $\mathbf{C} = \mathbf{KS}$ contains the columns of $\mathbf{K}$ indexed by $\mathcal{S} \subset [n]$, and the columns selected by $\mathbf{P}$ are indexed by $\mathcal{P} \subset [n]$. Empirically, enforcing $\mathcal{S} \subset \mathcal{P}$ significantly improves the approximation quality.

- Empirically, letting p be several times larger than s, e.g. $p = 4s$, is sufficient for a high quality.

The algorithm can be implemented in 12 lines of MATLAB code.

```
function [QC, Z] = spsdFaster(K, s)
p = 4 * s; % can be tuned
n = size(K, 2);
S = sort(randsample(n, s)); % uniform sampling
C = K(:, S);
[QC, ~] = qr(C, 0);
q = sum(QC.^2, 2); % the sampling probability
q = q / sum(q);
P = randsample(n, p,true, q); % leverage score sampling
P = unique([P; S]); % enforce P to contain S
PQCinv = pinv(QC(P, :));
Z = PQCinv * K(P, P) * PQCinv';
```

The above implementation assumes that $\mathbf{K}$ is a given matrix. In the kernel approximation problems, we are only given a $n \times d$ matrix $\mathbf{X}$, whose rows are data points, and a kernel function, e.g. the RBF kernel with width parameter σ. We should implement the faster SPSD sketching algorithm in the following way.

```
function [QC, Z] = spsdFaster(X, sigma, s)
p = 4 * s; % can be tuned
n = size(X, 1);
S = sort(randsample(n, s)); % uniform sampling
C = rbf(X, X(S, :), sigma);
[QC, ~] = qr(C, 0);
q = sum(QC.^2, 2); % the sampling probability
q = q / sum(q);
P = randsample(n, p,true, q);
P = unique([P; S]); % enforce P contains S
PQCinv = pinv(QC(P, :));
Ksub = rbf(X(P, :), X(P, :), sigma);
Z = PQCinv * Ksub * PQCinv';
```

The above implementation avoids computing the whole kernel matrix, and is thus highly efficient when applied to kernel methods.

6.4 The Nyström Method

Let $\mathbf{S}$ be an $n \times s$ column selection matrix and $\mathbf{C} = \mathbf{KS} \in \mathbb{R}^{n \times s}$ be a sketch of $\mathbf{K}$. Recall the model (6.2) proposed in the previous section. It is easy to verify that $\mathbf{Q}_\mathbf{C} \tilde{\mathbf{Z}} \mathbf{Q}_\mathbf{C}^T = \mathbf{C}\tilde{\mathbf{X}}\mathbf{C}^T$, where $\tilde{\mathbf{X}}$ is defined by

$$\tilde{\mathbf{X}} = \min_{\mathbf{X}} \|\mathbf{P}^T(\mathbf{K} - \mathbf{CXC})\mathbf{P}\|_F^2 = \underbrace{(\mathbf{P}^T\mathbf{C})^\dagger}_{s\times p} \underbrace{(\mathbf{P}^T\mathbf{KP})}_{p\times p} \underbrace{(\mathbf{C}^T\mathbf{P})^\dagger}_{p\times s}.$$

One can simply set $\mathbf{P} = \mathbf{S} \in \mathbb{R}^{n\times s}$ and let $\mathbf{W} = \mathbf{S}^T\mathbf{C} = \mathbf{S}^T\mathbf{KS}$. Then the solution $\tilde{\mathbf{X}}$ becomes

$$\tilde{\mathbf{X}} = (\mathbf{S}^T\mathbf{C})^\dagger(\mathbf{S}^T\mathbf{KS})(\mathbf{C}^T\mathbf{S})^\dagger = \mathbf{W}^\dagger\mathbf{W}\mathbf{W}^\dagger = \mathbf{W}^\dagger.$$

The low-rank approximation

$$\mathbf{K} \approx \mathbf{C}\mathbf{W}^\dagger\mathbf{C}^T$$

is called the Nyström method [20; 32]. The Nyström method is perhaps the most extensively used kernel approximation approach in the literature. See Figure 6.1 for the illustration of the Nyström method.

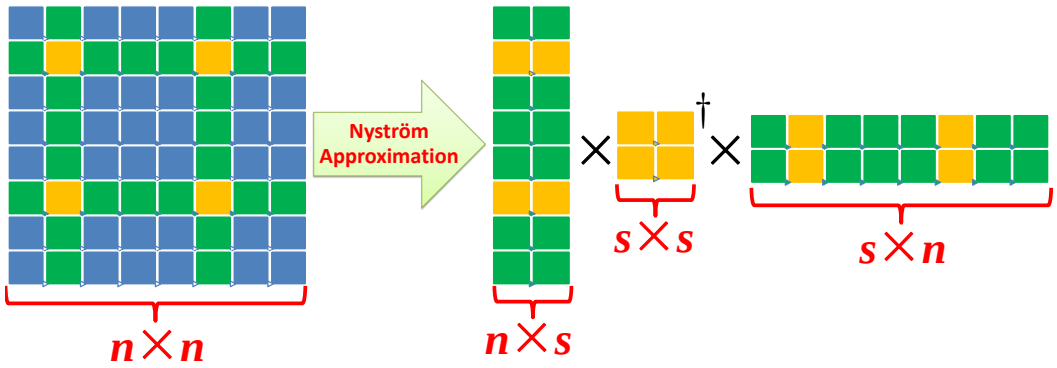


Figure 6.1: The illustration of the Nyström method.

There are a few things to remark:

- The Nyström is highly efficient. When applied to speedup kernel methods, the scalability can be as large as $n = 10^6$.
- The Nyström method is a rough approximation to $\mathbf{K}$ and is well known to be of low accuracy. If a moderately high accuracy is required, one had better use the method in the previous section.
- The $s \times s$ matrix $\mathbf{W}$ is usually ill-conditioned, and thus the Moore-Penrose inverse can be numerically instable. (It is because the bottom singular values of $\mathbf{W}$ blow up during the Moore-Penrose inverse.) A very effective heuristic is to drop the bottom singular values of $\mathbf{W}$: set a parameter $k < s$, e.g. $k = \lceil 0.8s \rceil$, and approximate $\mathbf{K}$ by $\mathbf{C}(\mathbf{W}_k)^\dagger\mathbf{C}^T$.

- There are many choices of the sampling matrix $\mathbf{S}$. See [13] for more discussions.

The Nyström method can be implemented in 11 lines of MATLAB code. The output of the algorithm is $\mathbf{L} \in \mathbb{R}^{n \times k}$, where $\mathbf{L}\mathbf{L}^T$ is the Nyström approximation to $\mathbf{K}$.[2]

```
1  function [L] = Nystrom(X, sigma, s)
2  k = ceil(0.8 * s); % can be tuned
3  n = size(X, 1);
4  S = sort(randsample(n, s)); % uniform sampling
5  C = rbf(X, X(S, :), sigma); % C = K(:, S)
6  W = C(S, :);
7  [UW, SW, ~] = svd(W);
8  SW = diag(SW);
9  SW = 1 ./ sqrt(SW(1:k));
10 UW = bsxfun(@times, UW(:, 1:k), SW');
11 L = C * UW; % K is approximated by L * L'
```

Here we use the RBF kernel function implemented in Section 6.1. Line 8 sets $k = \lceil 0.8c \rceil$, which can be better tuned to enhance numerical stability. Notice that k should not be set too small, otherwise the accuracy would be affected.

6.5 More Efficient Extensions

Several SPSD matrix approximation methods has been proposed recently, and they are more scalable than the Nyström method in certain applications. This section briefly describes some of these methods.

6.5.1 Memory Efficient Kernel Approximation (MEKA)

MEKA [24] exploits the block structure of kernel matrices and is more memory efficient than the Nyström method. MEKA first partitions the data $\mathbf{x}_1, \cdots, \mathbf{x}_n$ into b groups (e.g. by inexact $k$means clustering), accordingly, the kernel matrix $\mathbf{K}$ has $b \times b$ blocks:

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_{[1,1]} & \cdots & \mathbf{K}_{[1,b]} \\ \vdots & \ddots & \vdots \\ \mathbf{K}_{[b,1]} & \cdots & \mathbf{K}_{[b,b]} \end{bmatrix} = \begin{bmatrix} \mathbf{K}_{[1:]} \\ \vdots \\ \mathbf{K}_{[b:]} \end{bmatrix}.$$

Then MEKA approximately computes the top left singular vectors of $\mathbf{K}_{[1:]}, \cdots, \mathbf{K}_{[b:]}$, denote $\mathbf{U}_{[1]}, \cdots, \mathbf{U}_{[b]}$, respectively. For each $(i, j) \in [b] \times [b]$, MEKA finds a very small-scale matrix $\mathbf{Z}_{[i,j]}$ by solving

$$\mathbf{Z}_{[i,j]} = \operatorname*{argmin}_{\mathbf{Z}} \left\| \mathbf{K}_{[i,j]} - \mathbf{U}_{[i]} \mathbf{Z}_{[i,j]} \mathbf{U}_{[j]}^T \right\|_F^2.$$

[2] Let $\mathbf{W}_k = \mathbf{U}_{\mathbf{W},k} \mathbf{\Lambda}_{\mathbf{W},k} \mathbf{U}_{\mathbf{W},k}^T$ be the k-eigenvalue decomposition of $\mathbf{W}$ and set $\mathbf{L} = \mathbf{C}\mathbf{U}_{\mathbf{W},k} \mathbf{\Lambda}_{\mathbf{W},k}^{-1} \in \mathbb{R}^{n \times k}$.

This can be done efficiently using the approach in Section 4.5. Finally, the low-rank approximation is

$$\mathbf{K} \approx \begin{bmatrix} \mathbf{U}_{[1]} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathbf{U}_{[b]} \end{bmatrix} \begin{bmatrix} \mathbf{Z}_{[1,1]} & \cdots & \mathbf{Z}_{[1,b]} \\ \vdots & \ddots & \vdots \\ \mathbf{Z}_{[b,1]} & \cdots & \mathbf{Z}_{[b,b]} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{[1]} & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathbf{U}_{[b]} \end{bmatrix}^T = \mathbf{U}\mathbf{Z}\mathbf{U}^T.$$

Since $\mathbf{Z}$ and $\mathbf{U}_{[1]}, \cdots, \mathbf{U}_{[b]}$ are small-scale matrices, MEKA is thus very memory efficient. There are several things to remark:

- MEKA can be used to speedup Gaussian process regression and least squares SVM. However, MEKA can be hardly applied to speedup k-eigenvalue decomposition, because it requires the k-SVD of $\mathbf{U}\mathbf{Z}^{1/2}$, which destroys the sparsity and significantly increases memory cost.
- Indiscreet implementation, e.g. the implementation provided by [24], can make MEKA numerically unstable, as was reported by [30; 28]. The readers had better to follow the stabler implementation in [28].

6.5.2 Structured Kernel Interpolation (SKI)

SKI [33] is a memory efficient extension of the Nyström method. Let $\mathbf{S}$ be a column selection matrix, $\mathbf{C} = \mathbf{K}\mathbf{S}$, and $\mathbf{W} = \mathbf{S}^T\mathbf{C} = \mathbf{S}^T\mathbf{K}\mathbf{S}$. The Nyström method approximates $\mathbf{K}$ by $\mathbf{C}\mathbf{W}^\dagger\mathbf{C}^T$. SKI further approximates each row of $\mathbf{C}$ by a convex combination of two rows of $\mathbf{W}$ and obtain $\mathbf{C} \approx \mathbf{X}\mathbf{W}$. Notice that each row of $\mathbf{X}$ has only two nonzero entries, which makes $\mathbf{X}$ extremely sparse. In this way, $\mathbf{K}$ is approximated by

$$\mathbf{K} \approx \mathbf{C}\mathbf{W}^\dagger\mathbf{C} \approx (\mathbf{X}\mathbf{W})\mathbf{W}^\dagger(\mathbf{X}\mathbf{W})^T = \mathbf{X}\mathbf{W}\mathbf{X}^T.$$

Much accuracy is lost in the second approximation, so SKI is much less accurate than the Nyström method. For the same reason as MEKA, there is no point in applying SKI to speedup k-eigenvalue decomposition of $\mathbf{K}$.

6.6 Extension to Rectangular Matrices: CUR Matrix Decomposition

This section considers the problem of sketching any rectangular matrix $\mathbf{A}$ by the CUR matrix decomposition [16]. The CUR matrix decomposition is an extension of the previously discussed SPSD matrix sketching methods.

6.6.1 Motivation

Suppose we are given n training data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$, m test data $\mathbf{x}'_1, \cdots, \mathbf{x}'_m \in \mathbb{R}^d$, and a kernel function $\kappa(\cdot, \cdot)$. In their generalization (test) stage, kernel methods such as GPR and

KPCA form an $m \times n$ matrix $\mathbf{K}_*$, where $(\mathbf{K}_*)_{ij} = \kappa(\mathbf{x}'_i, \mathbf{x}_j)$, and apply $\mathbf{K}_*$ to some vectors or matrices. Notice that it takes $\mathcal{O}(mnd)$ time to form $\mathbf{K}_*$ and $\mathcal{O}(mnp)$ time to multiply $\mathbf{K}_*$ by an $n \times p$ matrix. If m is as large as n, the generalization stage of such kernel methods can be very expensive. Fortunately, with the help of the CUR matrix decomposition, the generalization stage of GPR or KPCA merely costs time linear in $m + n$.

6.6.2 Prototype CUR Decomposition

Suppose we are given an arbitrary $m \times n$ rectangular matrix $\mathbf{A}$, which can be the aforementioned $\mathbf{K}_*$. We sample c columns of $\mathbf{A}$ to form $\mathbf{C} = \mathbf{A}\mathbf{S}_\mathbf{C} \in \mathbb{R}^{m \times c}$, sample r rows of $\mathbf{A}$ to form $\mathbf{R} = \mathbf{A}\mathbf{S}_\mathbf{R} \in \mathbb{R}^{r \times n}$, and compute the intersection matrix $\mathbf{U}^\star \in \mathbb{R}^{c \times r}$ by solving

$$\mathbf{U}^\star \;=\; \operatorname*{argmin}_{\mathbf{U}} \| \underbrace{\mathbf{A}}_{m\times n} - \underbrace{\mathbf{C}}_{m\times c}\,\underbrace{\mathbf{U}}_{c\times r}\,\underbrace{\mathbf{R}}_{r\times n} \|_F^2 \;=\; \mathbf{C}^\dagger \mathbf{A} \mathbf{R}^\dagger. \tag{6.3}$$

The approximation $\mathbf{A} \approx \mathbf{C}\mathbf{U}^\star\mathbf{R}$ is well known as the CUR decomposition [16]. This formulation bears a strong resemblance with the prototype SPSD matrix sketching method in (6.1).

The prototype CUR decomposition is not very useful because (1) its time cost is $\mathcal{O}(mn \cdot \min\{c, r\})$ and (2) it visits every entry of $\mathbf{A}$.

6.6.3 Faster CUR Decomposition

Analogous to the SPSD matrix sketching, we can compute $\mathbf{U}^\star$ approximately and significantly more efficiently. Let $\mathbf{P}_\mathbf{C} \in \mathbb{R}^{m \times p_c}$ and $\mathbf{P}_\mathbf{R} \in \mathbb{R}^{n \times p_r}$ be some column selection matrices. Then we solve this problem in stead of (6.3):

$$\tilde{\mathbf{U}} \;=\; \operatorname*{argmin}_{\mathbf{U}} \| \underbrace{\mathbf{P}_\mathbf{C}^T\mathbf{A}\mathbf{P}_\mathbf{R}}_{p_c\times p_r} - \underbrace{\mathbf{P}_\mathbf{C}^T\mathbf{C}}_{p_c\times c}\,\underbrace{\mathbf{U}}_{c\times r}\,\underbrace{\mathbf{R}\mathbf{P}_\mathbf{R}}_{r\times p_r} \|_F^2 \;=\; (\mathbf{P}_\mathbf{C}^T\mathbf{C})^\dagger(\mathbf{P}_\mathbf{C}^T\mathbf{A}\mathbf{P}_\mathbf{R})(\mathbf{R}\mathbf{P}_\mathbf{R})^\dagger. \tag{6.4}$$

The faster CUR decomposition is very similar to the faster SPSD matrix sketching method in Section 6.3. The faster CUR decomposition has the following properties:

- It visits only $mc + nr + p_c p_r$ entries of $\mathbf{A}$, which is linear in $m + n$. This is particularly useful when applied to kernel methods, because it avoids forming the whole kernel matrix.
- The overall time and memory costs are linear in $m + n$.
- If $\mathbf{P}_\mathbf{C}$ is the leverage score sampling matrix corresponding to the columns of $\mathbf{C}^T$ and $\mathbf{P}_\mathbf{R}$ is the leverage score sampling matrix corresponding to the columns of $\mathbf{R}$, then $\tilde{\mathbf{U}}$ is a very high quality approximation to $\mathbf{U}^\star$ [31]:

  $$\|\mathbf{A} - \mathbf{C}\tilde{\mathbf{U}}\mathbf{R}\|_F^2 \;\le\; (1+\epsilon) \min_{\mathbf{U}} \|\mathbf{A} - \mathbf{C}\mathbf{U}\mathbf{R}\|_F^2$$

  holds with high probability.

Empirically speaking, setting $\mathbf{P_C}$ and $\mathbf{P_R}$ be uniform sampling matrices works nearly as well as leverage score sampling matrices, and setting $p_c = p_r = \mathcal{O}(c+r)$ suffices for a high approximation quality. If $\mathbf{A}$ is a full-observed matrix, the CUR matrix decomposition can be computed by the following MATLAB code.

```
function [C, U, R] = curFaster(A, c, r)
pc = 2 * (r + c); % can be tuned
pr = 2 * (r + c); % can be tuned
[m, n] = size(A);
SC = sort(randsample(n, c));
SR = sort(randsample(m, r));
C = A(:, SC);
R = A(SR, :);
PC = sort(randsample(m, pc));
PR = sort(randsample(n, pr));
PC = unique([PC; SR]); % enforce PC to contain SR
PR = unique([PR; SC]); % enforce PR to contain SC
U = pinv(C(PC, :)) * A(PC, PR) * pinv(R(:, PR));
```

Let's consider the kernel approximation problem in Section 6.6.1. Let $\mathbf{X}_{\text{train}} \in \mathbb{R}^{n \times d}$ be the training data and $\mathbf{X}_{\text{test}} \in \mathbb{R}^{m \times d}$ be the test data. We use the RBF kernel with kernel width parameter σ. The $m \times n$ matrix $\mathbf{K}_*$ can be approximated by $\tilde{\mathbf{K}}_* = \mathbf{CUR}$, which is the output of the following MATLAB procedure.

```
function [C, U, R] = curFasterKernel(Xtest, Xtrain, sigma, c, r)
pc = 2 * (r + c); % can be tuned
pr = 2 * (r + c); % can be tuned
m = size(Xtest, 1);
n = size(Xtrain, 1);
SC = sort(randsample(n, c));
SR = sort(randsample(m, r));
C = rbf(Xtest, Xtrain(SC, :), sigma);
R = rbf(Xtest(SR, :), Xtrain, sigma);
PC = sort(randsample(m, pc));
PR = sort(randsample(n, pr));
PC = unique([PC; SR]); % enforce PC to contain SR
PR = unique([PR; SC]); % enforce PR to contain SC
Kblock = rbf(Xtest(PC, :), Xtrain(PR, :), sigma);
U = pinv(C(PC, :)) * Kblock * pinv(R(:, PR));
```

The time cost of this procedure is linear in $m+n$, and $\tilde{\mathbf{K}}_* = \mathbf{CUR}$ can be applied to n dimensional vector in $\mathcal{O}\big(nr + mc\big)$ time. In this way, the generalization of GPR and KPCA can be efficient.

6.7 Applications

This section provides the implementations of kernel PCA, spectral clustering, Gaussian process regression, all sped-up by randomized algorithms.

6.7.1 Kernel Principal Component Analysis (KPCA)

Suppose we are given

- n training data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$,
- m test data $\mathbf{x}'_1, \cdots, \mathbf{x}'_m \in \mathbb{R}^d$, ($\mathbf{x}'_i$ is not the transpose $\mathbf{x}_i^T$),
- a kernel function $\kappa(\cdot, \cdot)$, e.g. the RBF kernel function,
- a target rank k ($\ll n, d$).

The goal of KPCA is to extract k features of each training datum and each test datum, which may be used in clustering or classification. The standard KPCA consists of the following steps:

1. Training
   (a) Form the $n \times n$ kernel matrix $\mathbf{K}$ of the training data, whose the (i, j)-th entry is $\kappa(\mathbf{x}_i, \mathbf{x}_j)$;
   (b) Compute the k-eigenvalue decomposition $\mathbf{K}_k = \mathbf{U}_k \mathbf{\Lambda}_k \mathbf{U}_k^T$;
   (c) Form the $n \times k$ matrix $\mathbf{U}_k \mathbf{\Lambda}_k^{1/2}$, whose the i-th row is the feature of $\mathbf{x}_i$;
2. Generalization (test)
   (a) Form the $m \times n$ kernel matrix $\mathbf{K}_*$ whose the (i, j)-th entry is $\kappa(\mathbf{x}'_i, \mathbf{x}_j)$;
   (b) Form the $m \times k$ matrix $\mathbf{K}_* \mathbf{U}_k \mathbf{\Lambda}_k^{-1/2}$, whose the i-th row is the feature of $\mathbf{x}'_i$.

The most time and memory expensive step in training is the k-eigenvalue decomposition of $\mathbf{K}$, which can be sped-up by the sketching techniques discussed in this section. Empirically, the faster SPSD matrix sketching in Section 6.3 is much more accurate than the Nyström method in Section 6.4, and their time and memory costs are all linear in n. Thus the faster SPSD matrix sketching can be better choice. KPCA can be approximately solved by several lines of MATLAB code.

```
function [U, lambda, featuretrain] = kpcaTrain(Xtrain, sigma, k)
s = k * 10; % can be tuned
[QC, Z] = spsdFaster(Xtrain, sigma, s); % QC has orthogonal columns
clear Xtrain
[UZ, SZ, ~] = svd(Z);
U = QC * UZ(:, 1:k); % U contains the top k eigenvectors
```

```
lambda = diag(SZ);
lambda = lambda(1:k); % lambda is the vector containing the top k
    eigenvalues
featuretrain = bsxfun(@times, U, (sqrt(lambda))');
end
```

```
function [featuretest] = kpcaTest(Xtrain, Xtest, sigma, U, lambda)
Ktest = rbf(Xtest, Xtrain, sigma);
U = bsxfun(@times, U, (1 ./ sqrt(lambda))');
featuretest = Ktest * U;
end
```

In the function "kpcaTrain", the input variable "Xtrain" has n rows, each of which corresponds to a training datum. The rows of the output "featuretrain" and "featuretest" are the features extracted by KPCA, and the features can be used to perform classification. For example, suppose each datum $\mathbf{x}_i$ is associated with a label y_i, and let $\mathbf{y} = [y_1, \cdots, y_n]^T \in \mathbb{R}^n$. We can use k-nearest-neighbor

```
[ytest] = knnclassify(featuretest, featuretrain, y)
```

to predict the labels of the test data.

When the number of test data m is large, the function "kpcaTest" is costly. The users should apply the CUR decomposition in Section 6.6.3 to speedup computation.

```
function [featuretest] = kpcaTestCUR(Xtrain, Xtest, sigma, U, lambda)
c = max(100, ceil(size(Xtrain, 1) / 20)); % can be tuned
r = max(100, ceil(size(Xtest, 1) / 20)); % can be tuned
[C, Utilde, R] = curFasterKernel(Xtest, Xtrain, sigma, c, r);
U = bsxfun(@times, U, (1 ./ sqrt(lambda))');
featuretest = C * (Utilde * (R * U));
end
```

6.7.2 Spectral Clustering

Spectral clustering is one of the most popular clustering algorithms. Suppose we are given

- n data points $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$,
- a kernel function $\kappa(\cdot, \cdot)$,
- k: the number of classes.

Spectral clustering performs the following operations:

1. Form an $n \times n$ kernel matrix $\mathbf{K}$, where big k_{ij} indicates $\mathbf{x}_i$ and $\mathbf{x}_j$ are similar;
2. Form the degree matrix $\mathbf{D}$ with $d_{ii} = \sum_j k_{ij}$ and $d_{ij} = 0$ for all $i \neq j$;

3. Compute the normalized graph Laplacian $\mathbf{G} = \mathbf{D}^{-1/2}\mathbf{K}\mathbf{D}^{-1/2} \in \mathbb{R}^{n\times n}$;

4. Compute the top k eigenvectors of $\mathbf{G}$, denote $\mathbf{U} \in \mathbb{R}^{n\times k}$, and normalize the rows of $\mathbf{U}$;

5. Apply $k$means clustering on the rows of $\mathbf{V}$ to obtain the class labels.

The first step costs $\mathcal{O}(n^2 d)$ time and the fourth step costs $\mathcal{O}(n^2 k)$ times, which limit the scalability of spectral clustering. Fowlkes *et al.* [11] proposed to apply the Nyström method to make spectral clustering more scalable by avoiding forming the whole kernel matrix and speeding-up the k-eigenvalue decomposition. Empirically, the algorithm in Section 6.3 is more accurate than the Nyström method in Section 6.4, and they both runs in linear time. Spectral clustering with the randomized algorithm in Section 6.3 can be implemented in 16 lines of MATLAB code.

```
1  function [labels] = SpectralClusteringFaster(X, sigma, k)
2  s = k * 10; % can be tuned
3  n = size(X, 1);
4  [QC, Z] = spsdFaster(X, sigma, s); % K is approximated by QC * Z * QC'
5  [UZ, SZ, ~] = svd(Z);
6  SZ = sqrt(diag(SZ));
7  UZ = bsxfun(@times, UZ, SZ'); % now Z = UZ * UZ'
8  L = QC * UZ; % now K is approximated by L * L'
9  d = ones(n, 1);
10 d = L * (L' * d); % diagonal of the degree matrix D
11 d = 1 ./ sqrt(d);
12 L = bsxfun(@times, L, d); % now G is approximated by L*L'
13 [U, ~, ~] = svd(L, 'econ');
14 U = U(:, 1:k);
15 U = normr(U); % normalize the rows of U
16 labels = kmeans(U, k, 'Replicates', 3);
```

When the scale of data is too large for the faster SPSD matrix sketching algorithm in Section 6.3, one can instead use the more efficient Nyström method in Section 6.4: simply replace Lines 4 to 8 by

```
1  L = Nystrom(X, sigma, s);
```

6.7.3 Gaussian Process Regression (GPR)

The Gaussian process regression (GPR) is one of the most popular machine learning methods. GPR is the foundation of Bayesian optimization and has important applications such as automatically tuning the hyper-parameters of deep neural networks. Suppose we are given

- n training data $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbf{R}^d$,
- labels $\mathbf{y} = [y_1, \cdots y_n]^T \in \mathbb{R}^n$ of the training data,
- m test data $\mathbf{x}'_1, \cdots, \mathbf{x}'_m \in \mathbb{R}^d$, ($\mathbf{x}'_i$ is not the transpose $\mathbf{x}_i^T$),

- and a kernel function $\kappa(\cdot,\cdot)$, e.g. the RBF kernel with kernel width parameter σ.

Training. In the training stage, GPR requires forming the $n \times n$ kernel matrix $\mathbf{K}$ where $k_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ and computing the model

$$\mathbf{w} \;=\; (\mathbf{K} + \alpha \mathbf{I}_n)^{-1}\mathbf{y}.$$

Here α is a tuning parameter that indicates the noise intensity in the labels $\mathbf{y}$. It takes $\mathcal{O}(n^2 d)$ time to form the kernel matrix and $\mathcal{O}(n^3)$ time to compute the matrix inversion. To make the training efficient, we can first sketch the SPSD matrix $\mathbf{K}$ to obtain $\mathbf{K} \approx \mathbf{L}\mathbf{L}^T$ and then apply the technique in Section 6.1.2 to obtain $\mathbf{w}$. Empirically, when applied to speedup GPR, the algorithms discussed in Section 6.3 and Section 6.4 has similar accuracy, thus we choose to use the Nyström method which is more efficient.

The training GPR with the Nyström approximation can be implemented in the following MATLAB code. The time cost is $\mathcal{O}(nl^2 + nld)$ and the space cost is $\mathcal{O}(nl + nd)$.

```
function [w] = gprTrain(Xtrain, ytrain, sigma, alpha)
l = 100; % can be tuned
L = Nystrom(Xtrain, sigma, l); % K is approximated by L * L'
l = size(L, 2);
w = L' * ytrain;
w = (alpha * eye(l) + L' * L) \ w;
w = ytrain - L * w;
w = w / alpha;
end
```

The input "sigma" is the kernel width parameter and "alpha" indicates the noise intensity in the observation.

Generalization (test). After obtaining the trained model $\mathbf{w} \in \mathbb{R}^n$, GPR can predict the unknown labels of the m test data $\mathbf{x}'_1, \cdots, \mathbf{x}'_m \in \mathbb{R}^d$. GPR forms an $m \times n$ kernel matrix $\mathbf{K}_*$ whose the (i,j)-th entry is $\kappa(\mathbf{x}'_i, \mathbf{x}_j)$ and compute $\mathbf{y}_* = \mathbf{K}_*\mathbf{w} \in \mathbb{R}^m$. The i-th entry in $\mathbf{y}_*$ is the predictive label of $\mathbf{x}'_i$. The generalization can be implemented in four lines of MATLAB code.

```
function [ytest] = gprTest(Xtrain, Xtest, sigma, w)
Ktest = rbf(Xtest, Xtrain, sigma);
ytest = Ktest * w;
end
```

It costs $\mathcal{O}(mnd)$ time to compute $\mathbf{K}_*$ and $\mathcal{O}(mn)$ time to apply $\mathbf{K}_*$ to $\mathbf{w}$. If m is small, the generalization stage can be performed straightforwardly. However, if m is as large as n, the time cost will be quadratic in n, and the user should apply the CUR decomposition in Section 6.6.3 to speedup computation.

```
function [ytest] = gprTestCUR(Xtrain, Xtest, sigma, w)
c = max(100, ceil(size(Xtrain, 1) / 20)); % can be tuned
r = max(100, ceil(size(Xtest, 1) / 20)); % can be tuned
[C, Utilde, R] = curFasterKernel(Xtest, Xtrain, sigma, c, r);
ytest = C * (Utilde * (R * w));
end
```

Appendix A

Several Facts of Matrix Algebra

This chapter lists some facts that has been applied in this paper.

Fact A.1. *The matrices* $\mathbf{Q}_1 \in \mathbb{R}^{m\times n}$ *and* $\mathbf{Q}_{n\times p}$ *($m \geq n \geq p$) have orthonormal columns. Then the matrix* $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2$ *has orthonormal columns.*

Fact A.2. *Let* $\mathbf{A}$ *be any* $m\times n$ *and rank* ρ *matrix. Then* $\mathbf{A}\mathbf{A}^\dagger\mathbf{B} = \mathbf{U}_\mathbf{A}\mathbf{U}_\mathbf{A}^T\mathbf{B} = \mathbf{A}\mathbf{X}^\star = \mathbf{U}_\mathbf{A}\mathbf{Z}^\star$, *where*

$$\mathbf{X}^\star \;=\; \underset{\mathbf{X}}{\operatorname{argmin}} \|\mathbf{B}-\mathbf{A}\mathbf{X}\|_F^2, \qquad \textit{and} \qquad \mathbf{Z}^\star \;=\; \underset{\mathbf{Z}}{\operatorname{argmin}} \|\mathbf{B}-\mathbf{U}_\mathbf{A}\mathbf{Z}\|_F^2.$$

This is the reason why $\mathbf{A}\mathbf{A}^\dagger\mathbf{B}$ *and* $\mathbf{U}_\mathbf{A}\mathbf{U}_\mathbf{A}^T\mathbf{B}$ *are called the projection of* $\mathbf{B}$ *onto the column space of* $\mathbf{A}$.

Fact A.3. *[34, Lemma 44] The matrices* $\mathbf{Q} \in \mathbb{R}^{m\times s}$ *($m \geq s$) has orthonormal columns. The solution to*

$$\underset{\operatorname{rank}(\mathbf{X})\leq k}{\operatorname{argmin}} \|\mathbf{A}-\mathbf{Q}\mathbf{X}\|_F^2$$

is $\mathbf{X}^\star = (\mathbf{Q}^T\mathbf{A})_k$, *where* $(\mathbf{Q}^T\mathbf{A})_k$ *denotes the closest rank* k *approximation to* $\mathbf{Q}^T\mathbf{A}$.

Fact A.4. *Let* $\mathbf{A}^\dagger$ *be the Moore-Penrose inverse of* $\mathbf{A}$. *Then* $\mathbf{A}\mathbf{A}^\dagger\mathbf{A} = \mathbf{A}$ *and* $\mathbf{A}^\dagger\mathbf{A}\mathbf{A}^\dagger = \mathbf{A}^\dagger$.

Fact A.5. *Let* $\mathbf{A}$ *be an* $m\times n$ *($m \geq n$) matrix and* $\mathbf{A} = \mathbf{Q}_\mathbf{A}\mathbf{R}_\mathbf{A}$ *be the QR decomposition of* $\mathbf{A}$. *Then*

$$\underbrace{\mathbf{A}^\dagger}_{n\times m} \;=\; \underbrace{\mathbf{R}_\mathbf{A}^\dagger}_{n\times n}\underbrace{\mathbf{Q}_\mathbf{A}^T}_{n\times m}.$$

Fact A.6. *Let* $\mathbf{C}$ *be a full-rank matrix with more rows than columns. Let* $\mathbf{C} = \mathbf{Q}_\mathbf{C}\mathbf{R}_\mathbf{C}$ *be the QR decomposition and* $\mathbf{C} = \mathbf{U}_\mathbf{C}\mathbf{\Sigma}_\mathbf{C}\mathbf{V}_\mathbf{C}$ *be the condensed SVD. Then the leverage scores of* $\mathbf{C}$, $\mathbf{Q}_\mathbf{C}$, $\mathbf{U}_\mathbf{C}$ *are the same.*

Appendix B

Notes and Further Reading

The ℓ_p Regression Problems. Chapter 4 has applied the sketching methods to solve the ℓ_2 norm regression problem more efficiently. The more general ℓ_p regression problems have also been studied in the literature [5; 7; 17; 8]. Especially, the ℓ_1 is of great interest because it demonstrate strong robustness to noise. Currently the strongest result is the ℓ_p row sampling by Lewis weights [8].

Distributed SVD. In the distributed model, each machine holds a subset of columns of $\mathbf{A}$, and the system outputs the top singular values and singular vectors. In this model, the communication cost should also be considered, as well as the time and memory costs. The seminal work [10] proposed to build a coreset to capture the properties of $\mathbf{A}$, which facilitates low computation and communication costs. Later on, several algorithms with stronger error bound and lower communication and computation costs have been proposed. Currently, the state of the art is [2].

Random Feature for Kernel Methods. Chapter 6 has introduced the sketching methods for kernel methods. A parallel line of work is the random feature methods [22] which also form low-rank approximations to kernel matrices. Section 6.5.3 of [27] offers simple and elegant proof of a random feature method. Since the sketching methods usually works better than the random feature methods (see the examples in [35]), the users are advised to apply the sketching methods introduced in Chapter 6. Besides the two kinds of low-rank approximation approaches, the stochastic optimization approach [9] also demonstrates very high scalability.

Bibliography

[1] Adi Ben-Israel and Thomas N.E. Greville. *Generalized Inverses: Theory and Applications. Second Edition*. Springer, 2003.

[2] Christos Boutsidis and David P Woodruff. Communication-optimal distributed principal component analysis in the column-partition model. *arXiv preprint arXiv:1504.06729*, 2015.

[3] Christos Boutsidis, Petros Drineas, and Malik Magdon-Ismail. Near-optimal column-based matrix reconstruction. *SIAM Journal on Computing*, 43(2):687–717, 2014.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[5] Kenneth L Clarkson. Subgradient and sampling algorithms for l1 regression. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 257–266. Society for Industrial and Applied Mathematics, 2005.

[6] Kenneth L. Clarkson and David P. Woodruff. Low rank approximation and regression in input sparsity time. In *Annual ACM Symposium on theory of computing (STOC)*. ACM, 2013.

[7] Kenneth L Clarkson, Petros Drineas, Malik Magdon-Ismail, Michael W Mahoney, Xiangrui Meng, and David P Woodruff. The fast cauchy transform and faster robust linear regression. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 466–477. SIAM, 2013.

[8] Michael B Cohen and Richard Peng. ℓ_p row sampling by lewis weights. *arXiv preprint arXiv:1412.0588*, 2014.

[9] Bo Dai, Bo Xie, Niao He, Yingyu Liang, Anant Raj, Maria-Florina F Balcan, and Le Song. Scalable kernel methods via doubly stochastic gradients. In *Advances in Neural Information Processing Systems (NIPS)*. 2014.

[10] Dan Feldman, Melanie Schmidt, and Christian Sohler. Turning big data into tiny data: Constant-size coresets for k-means, pca and projective clustering. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1434–1453. SIAM, 2013.

[11] Charless Fowlkes, Serge Belongie, Fan Chung, and Jitendra Malik. Spectral grouping using the Nyström method. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):214–225, 2004.

[12] Alex Gittens. The spectral norm error of the naive Nyström extension. *arXiv preprint arXiv:1110.5305*, 2011.

[13] Alex Gittens and Michael W. Mahoney. Revisiting the nyström method for improved large-scale machine learning. In *International Conference on Machine Learning (ICML)*, 2013.

[14] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[15] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics*, 26(189-206), 1984.

[16] Michael W. Mahoney. Randomized algorithms for matrices and data. *Foundations and Trends in Machine Learning*, 3(2):123–224, 2011.

[17] Xiangrui Meng and Michael W Mahoney. Low-distortion subspace embeddings in input-sparsity time and applications to robust linear regression. In *Proceedings of the forty-fifth annual ACM symposium on theory of computing*, pages 91–100. ACM, 2013.

[18] Xiangrui Meng, Michael A Saunders, and Michael W Mahoney. Lsrn: A parallel iterative solver for strongly over-or underdetermined systems. *SIAM Journal on Scientific Computing*, 36(2):C95–C118, 2014.

[19] Cameron Musco and Christopher Musco. Stronger approximate singular value decomposition via the block Lanczos and power methods. *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[20] Evert J. Nyström. Über die praktische auflösung von integralgleichungen mit anwendungen auf randwertaufgaben. *Acta Mathematica*, 54(1):185–204, 1930.

[21] Ninh Pham and Rasmus Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *the 19th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 239–247. ACM, 2013.

[22] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems (NIPS)*, pages 1177–1184, 2007.

[23] Yousef Saad. Numerical methods for large eigenvalue problems. *preparation. Available from: http://www-users. cs. umn. edu/saad/books. html*, 2011.

[24] Si Si, Cho-Jui Hsieh, and Inderjit Dhillon. Memory efficient kernel approximation. In *International Conference on Machine Learning (ICML)*, pages 701–709, 2014.

[25] G. W. Stewart. Four algorithms for the efficient computation of truncated pivoted QR approximations to a sparse matrix. *Numerische Mathematik*, 83(2):313–323, 1999.

[26] Mikkel Thorup and Yin Zhang. Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. *SIAM J. Comput.*, 41(2): 293–331, April 2012. ISSN 0097-5397.

[27] Joel A Tropp. An introduction to matrix concentration inequalities. *arXiv preprint arXiv:1501.01571*, 2015.

[28] Ruoxi Wang, Yingzhou Li, Michael W Mahoney, and Eric Darve. Structured block basis factorization for scalable kernel matrix evaluation. *arXiv preprint arXiv:1505.00398*, 2015.

[29] Shusen Wang and Zhihua Zhang. Improving CUR matrix decomposition and the Nyström approximation via adaptive sampling. *Journal of Machine Learning Research*, 14:2729–2769, 2013.

[30] Shusen Wang, Luo Luo, and Zhihua Zhang. Spsd matrix approximation via column selection: Theories, algorithms, and extensions. *CoRR*, abs/1406.5675, 2014.

[31] Shusen Wang, Zhihua Zhang, and Tong Zhang. Towards more efficient symmetric matrix sketching and CUR matrix decomposition. *arXiv preprint arXiv:1503.08395*, 2015.

[32] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Advances in Neural Information Processing Systems (NIPS)*, 2001.

[33] Andrew Gordon Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). *arXiv preprint arXiv:1503.01057*, 2015.

[34] David P Woodruff. Sketching as a tool for numerical linear algebra. *arXiv preprint arXiv:1411.4357*, 2014.

[35] Tianbao Yang, Yu-Feng Li, Mehrdad Mahdavi, Rong Jin, and Zhi-Hua Zhou. Nyström method vs random fourier features: A theoretical and empirical comparison. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[36] Kai Zhang and James T. Kwok. Clustered Nyström method for large scale manifold learning and dimension reduction. *IEEE Transactions on Neural Networks*, 21(10): 1576–1587, 2010.